Consistent Data Aggregate Retrieval for Sensor Network Systems

LEE Lok Hang

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

July 2005


香港中文大學
大學
圖書館系統
18 NOV 2006
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Abstract of thesis entitled:

Consistent Data Aggregate Retrieval for Sensor Network Systems

Submitted by LEE Lok Hang

for the degree of Master of Philosophy

at Computer Science and Engineering in July 2005

Data retrieval in sensor networks is a broad field, and the retrieval of data aggregates is one of the most important topics. It is because data aggregates, instead of individual sensor values, are more descriptive and useful.

In the literature, several data aggregation algorithms aiming at the retrieval of data aggregates from a single set of sensors in a sensor network have been proposed. However, when data aggregates of several sets of sensors are needed, the only solution provided by these techniques is to build multiple distributed data structures or gossip groups in the sensor network. Hence in a sensor network containing N sensors, we may need 2^N distributed data structures or gossip groups in order to retrieve the aggregates from all possible sets of sensors.

In this thesis, we propose to build distributed data cubes for the fast retrieval of aggregate sums and aggregate averages from multiple regions in a sensor network, such that only one distributed data structure is needed. The proposed distributed data cube construction algorithms are based on the inclusion-exclusion principle, and they are capable of building distributed prefix sum and local prefix sum data cubes in sensor networks.

We also study data consistency in sensor networks, and we will propose a mechanism for the proposed distributed data cubes so that they can be constructed, updated, and queried consistently.

感測器網路系統的一致性資料彙總檢索

作者　李樂恆

香港中文大學計算機科學及工程學系

二零零五年七月

摘要

感測器網路資料檢索是一個寬廣的領域，而資料彙總檢索更是其中最重要的一個題目，因爲資料彙總比個別感測器的數據更能描述整個感測器網路的狀況。

在過去的研究裏，學者提出了幾種針對感測器網路中單一區域的資料彙總算法。可是，當我們想得到超過一個區域的資料彙總，這些現有技術所提供的唯一解決方法是在同一感測器網路內建立多個分佈式的資料結構或密語小組。因此在包含了 N 個感測器的感測器網路，我們需要多達 2^N 個分佈戎的資料結構或密語小組以得到所有區域的資料彙總。

在這份論文裏，我們提議透過建立分佈式的資料立方體以快速地從一個感測器網路的多個區域中得到它們的彙總和及彙總平均，而這種方法只需要一個分佈式的資料結構。我們提出的分佈式資料立方體建築算法建基於排容原理，並且能夠在感測器網路上建立分佈式前置和及局部前置和資料立方體。

我們亦研究了感測器網路的資料一致性，並提出一套簡單的方案令分佈式資料立方體可以一致地被建立、被更新及被查詢。

Acknowledgement

I still remember how untouchable M.Phil. was to me when I was in my undergraduate study. After knowing the academic result at the end of the second semester in Year 1, I was so despair and I believed that studying M.Phil. was just a dream. Now when I talk to my friends, I still regard the event, that I was selected to be a M.Phil. student by The Chinese University of Hong Kong, as a miracle. Fortunately, it is not a mirage and I am about to arrive at the finish point of the M.Phil. programme. Hence I would like to express my thanks to those people who have given me their help and care.

First of all, I would like to thank my supervisor Prof. WONG Man Hon for his guidance and advice during my M.Phil. study. Prof. WONG's help is very important to me, especially at the beginning of my M.Phil. study. At that time, I felt lost and I did not really know how to find a problem to solve. Prof. WONG showed me his patience and let me know how to dig out a problem.

Besides Professor WONG, let me thank my markers Professor FU Wai Chee and Professor WONG Tien Tsin. Thank you very much for spending invaluable time on my thesis and oral defence. I would also wish to thank my external marker.

Professor YOUNG Fung Yu is another professor that I should thank. Before I was selected to be an M.Phil. student, I sometimes dropped by her office and asked for her advice. She is really nice, so that she was still willing to chat with me even when she was busy.

I am indebted to my friends from St. Jude Catholic Church. They include Agnes, Bernard, Diana, Dominic, Edmond, Irene, Iris, Joanna, John, Kenneth, Ping, Samantha, Shirley SEE, and other sisters and brothers. They support me spiritually and mentally.

Thank Albert, Alex, Edith, Jill, Oscar, Pat, Royce and Shirley NG. They are my classmates of my M.Phil. study. Thank them for being with me everyday. I will always remember the laughters, the shouts, and the jokes we made in the last two years.

I would also want to thank Idy. She was always available to me when I wanted to share with her the problems I encountered in my research, though she never understands those technical things. She is so caring and loving, such that she can turn my every problem to happiness.

I must also give my profoundest thanks to my family, especially my mother. Mother, I really thank you very much.

This work is dedicated to God,

and my beloved mother.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reducing the size of computers while enriching their capabilities has long been one of the goals of computer scientists and computer engineers. As technology advances, nowadays we are able to produce computers which are so tiny that they can be embedded in sensor devices to form sensor nodes. A sensor node is a complex of sensing devices, processor, memory, and radio-frequency signal emitter and receiver. Sensor nodes can monitor the surrounding environment and generate environmental data continuously, as long as they do not run out of batteries. They are also capable of exchanging data through radio-frequency wireless channels. Therefore, they can be used for the collection of data on temperature, radioactivity, road traffic, battlefield surveillance, and stock inventory etc., which can then be further analyzed for useful information.

Sensor networks are one category of distributed systems. They are characterized by being highly distributed, dynamic, and wireless in nature. A sensor network is a network of sensor nodes which are small, cheap, and self-configurable, and a sensor network usually contains a large number of sensors [1]. Sensor networks can be deployed easily by sprinkling sensors from an aeroplane, and the sensor nodes will form a network autonomously. Once a sensor network is formed, the region covered by the sensor network can be monitored continuously and the data generated by the sensors can be collected from base stations.

Since sensor nodes are small and wireless, they have their physical limitations. For example, they have limited power supply, memory, and communication bandwidth. Despite these physical limitations, they have a large variety of functions. For example, they can measure the temperature, humidity, light intensity, and radioactivity of the surrounding environment. Some real-life sensor network applications and projects include the Environment Observation and Forecasting System (EOFS) [2] (e.g. CORIE [3] and ALERT [4]), habitat monitoring systems [5] (e.g. Habitat Monitoring on Great Duck Island [6]), traffic control systems [7] and fire detection systems [8].

In this thesis, we put our focus on the retrieval of data aggregates from sensor networks. It is because a sensor network may contain thousands of sensor nodes [1] and individual sensor values cannot describe the status of the whole network, and hence people seldom ask for the values of some individual sensors in a sensor network. Actually many researchers also agree that data aggregates are more useful than individual sensor values in sensor network systems [9–11].

The objective of data aggregation in sensor networks is to obtain the readings from all or some of the sensor nodes in a sensor network, accompanied by a common operation like SUM, AVG, MAX, and MIN. A successful solution to this problem must be scalable, able to reduce network traffic as well as sensor energy consumption, and is adaptable to different network topologies. In the literature, three kinds of data aggregation techniques have been proposed. They are hierarchical data aggregation model [10,12], gossip-based aggregation model [13], and a mixture of the two [14, 15]. These solutions are capable of calculating the data aggregates of a single region in a sensor network. However, they are not feasible when we want to retrieve data aggregates from more than one set of sensors. Consider a sensor network with N sensor nodes. If we apply existing aggregation techniques to retrieve the data aggregates from all of the 2^N possible sets of sensors in the sensor network, in the worst case

we need to build an aggregation tree or gossip group in each of the sets. As a result, we will have up to 2^N trees or gossip groups in the sensor network.

To solve the above problem, we propose to build distributed data cubes in sensor networks. With the two proposed distributed data cubes, the retrieval of aggregate sums and aggregate averages from multiple regions in a sensor network can be performed simultaneously with a constant number of operations.

In addition to the retrieval of data aggregates, we also study data consistency in sensor networks in the latter half of this thesis. We will propose a simple synchronous protocol for the two proposed distributed data cubes, so that they can be constructed, updated, and queried consistently.

1.1 Sensors and Sensor Networks

The rapid advance of Micro-electro-mechanical systems (MEMS) allows tiny sensing devices with wireless communication capabilities to be produced at low costs. Therefore, large scale wireless distributed networks with sensing abilities can be densely and widely deployed.

Sensor networks are a new technology, and the present status of sensor networks is similar to that of the Internet thirty years ago [16]. Therefore new algorithms still keep emerging to fulfill the requirements of different applications.

Like many other technologies, the birth of sensor nodes was due to military purpose. In 1990s, the Pentagon began a project for the development of smart sensor nodes for tracking enemies and enemy vehicles. After signing a contract with the Pentagon, the University of California Berkeley (UCB) then invented sensor nodes in the late 1990s. Gradually the technology was migrated to non-military areas, and until now sensor nodes and sensor networks attract a lot of researchers.

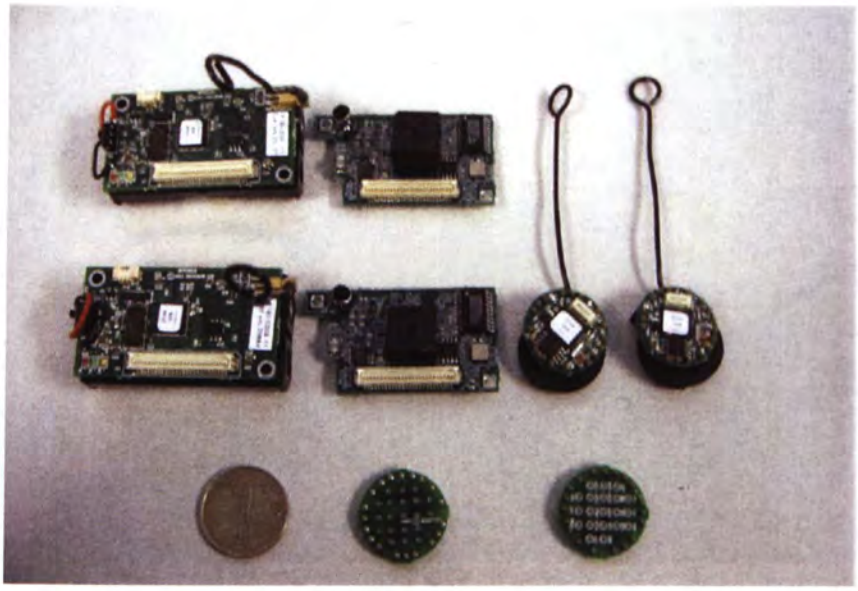
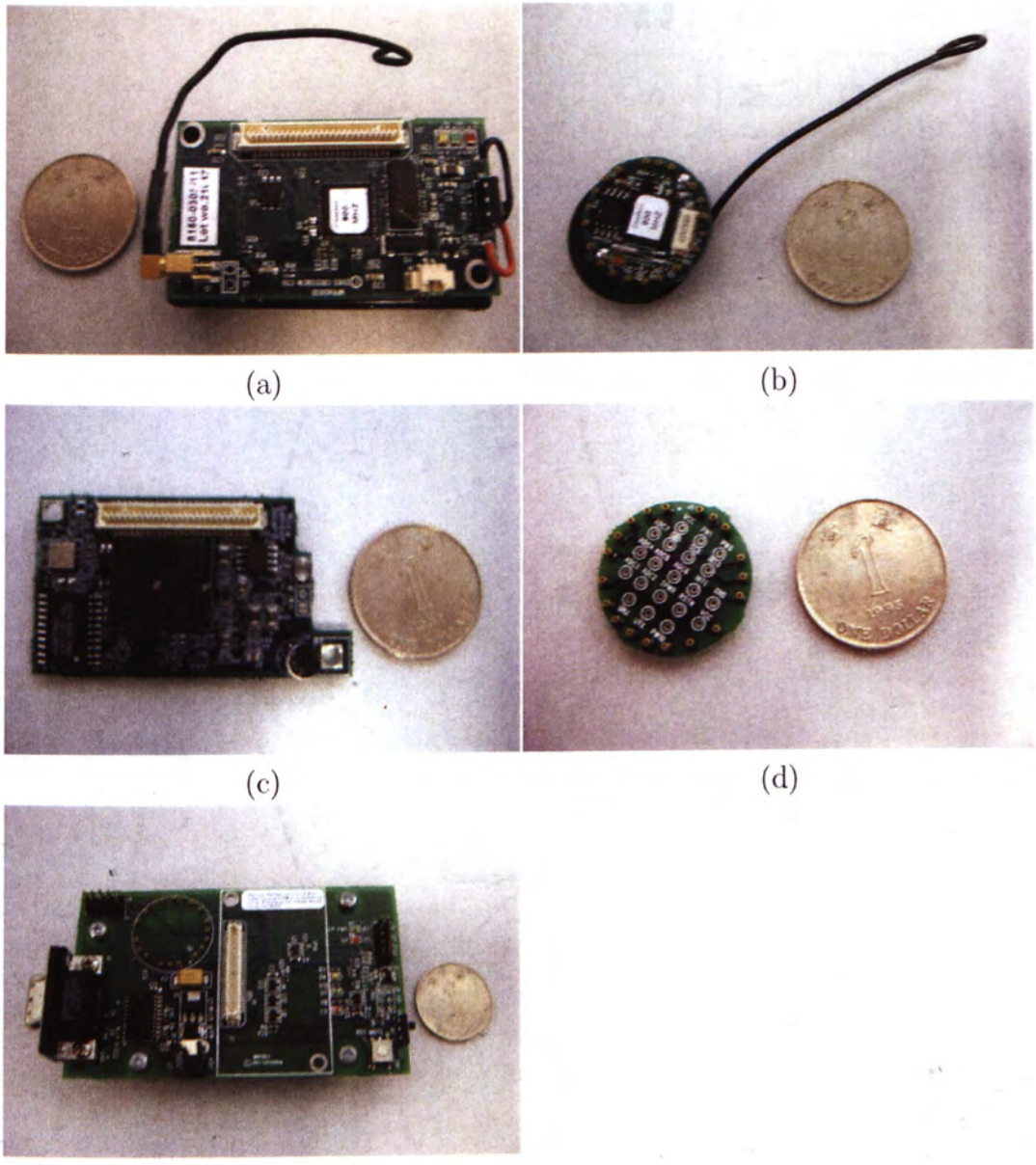


Figure 1.1: MOTE-KIT 5040 - MICA2/MICA2DOT Professional Kit

To satisfy different requirements and conditions, several sensor prototypes have been developed. Among them, UCB motes [17, 18] developed by UCB are the most famous ones. Other available prototypes include uAMPS [19], GNOMES [20], and PC104 [21]. On the other hand, some commercial sensor products are available now and Crossbow's UCB based motes [22] are the most popular.

Fig.1.1 shows the Crossbow mote kit that we used for the evaluation of the sensor technology. Its model number is MOTE-KIT5040, and it contains the following components:

- 4 MICA2 Processor/Radio Boards (Fig.1.2(a))
- 4 MICA2DOT Quarter-Sized Processor/Radio Boards (Fig.1.2(b))
- 3 MTS310 Sensor Boards (Acceleration, Magnetic, Light, Temperature, Acoustic, and Sounder)(Fig.1.2(c))
- 2 MDA500, MICA2DOT Prototype and Data Acquisition Boards (Fig.1.2(d))
- 1 MIB510 Programming and Serial Interface Board (Fig.1.2(e))



(a) (b)

(c) (d)

(e)

Figure 1.2: The components of MOTE-KIT5040

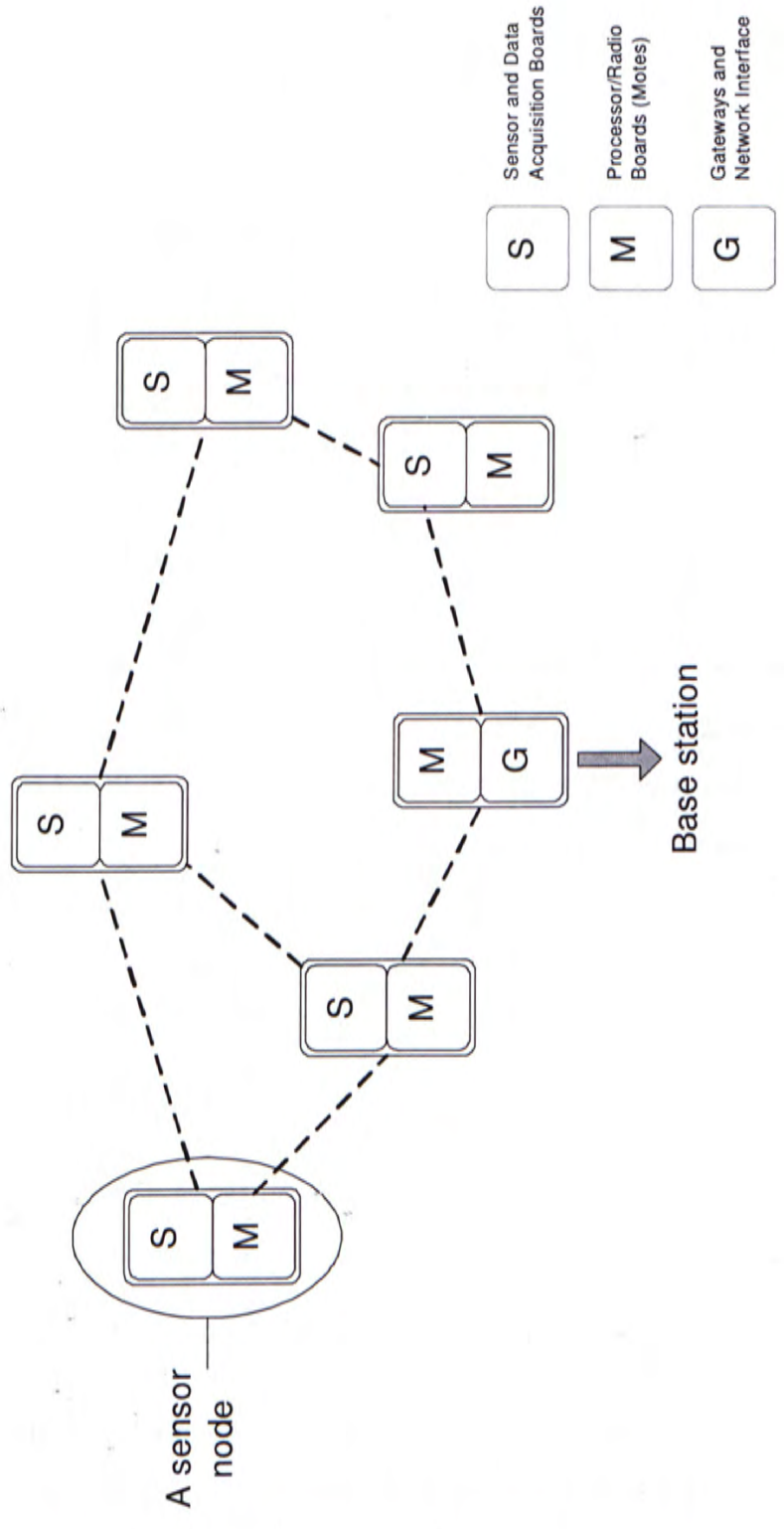


Figure 1.3: How a sensor network can be deployed

Notice that the frequency of the communication channel of the sensors we own is 900MHz, and the radio range is around $30m$ to $100m$. Finally, an operating system for sensor nodes is embedded into each Crossbow processor board, and it is called TinyOS.

1.2 Sensor Network Deployment

Fig.1.3 shows how a sensor network can be deployed with our sensor equipment, we can see that the mote kit contains three kinds of components: sensor and data acquisition boards (Fig.1.2(c) and Fig.1.2(d)), processor/radio boards (Fig.1.2(a) and Fig.1.2(b)), and gateways and network interface Fig.1.2(e). From the figure, a sensor node refers to the combination of a sensor and data acquisition board and a processor/radio board. To deploy a sensor network using Crossbow motes, we need to distribute a number of sensor nodes. After that, the sensors will work autonomously to form a sensor network. To collect data from sensors for further processing, we can set up a gateway using the combination of a processor/radio board and a gateway and network interface. Then we can communicate with the sensors using a PC or the Internet.

1.3 Motivations

People set up sensor networks because they want to monitor different environments and obtain environmental data over a period of time. Let us classify the data returned by a sensor network into two different kinds: individual sensor value and the data aggregate of a set of sensors. An example of individual sensor value is the temperature measured by sensor S107 in Lecture Room 508, and the value returned by S107 just refers to the temperature of a very small region in Lecture Room 508. On the other hand, the average value of the temperatures measured by all sensors in Lecture Room 508 would then be

a data aggregate.

If we adjust the room temperature of Lecture Room 508 based on the value returned by S107, then we may receive a mountain of complaint letters. It is because the temperature returned by S107 cannot tell us the exact overall temperature of Lecture Room 508. If S107 is next to a lamp, then the temperature measured by S107 may be higher than the actual average value. On the other hand, if S107 is close to the vent of the air conditioner, then the value returned would be lower than the average value. To control the room temperature of Lecture Room 508, we certainly need the average value returned by all sensors in the lecture room (given that the sensors are more or less evenly distributed). From this example, we can know that data aggregates are more meaningful than individual sensor values in sensor network systems as stated in [10, 11]. That is why we focus on the retrieval of data aggregates in this thesis.

Though several data aggregation techniques for sensor networks have been proposed [10, 12–15], we still insist to develop a new kind of data aggregation because existing techniques work well only when we want to get the data aggregate of a single region in a sensor network. Let us consider this scenario: a road traffic monitoring system is developed in a very crowded area of a city, and one of the data aggregation techniques mentioned above is implemented in the system. Therefore, we are able to obtain the average, maximum, and minimum traffic flow of that busy region. However, what can we do if we want to know the statistics of all the roads of that region? We need to modify the system so that a distributed data structure is built for each road using the sensors of that road, or we need to construct a gossip group for each road. On another day, if we want to know the average traffic of each section of each road, we need to modify the system again and build more distributed data structures or gossip groups. Obviously the system is not a feasible one, and hence we propose to construct distributed data cubes in sensor networks so that you can query for the data aggregates of any region in a sensor network using only one

distributed data structure. In other word, in a sensor network with N sensors, theoretically we can get the data aggregates of all the 2^N regions in the sensor network with only one distributed data structure.

The idea of building distributed data cubes in sensor networks is inspired by the use of data cubes [9,23–27] in traditional database systems. Prefix sum data cube [23] and local prefix sum data cube [26, 28] are two of the most fundamental data cubes, and we propose to apply them in sensor networks in the form of distributed data structures.

In the second half of this thesis, we will introduce a simple synchronous algorithm for the two proposed distributed data cubes, such that they can be constructed, updated, and queried consistently.

1.4 Contributions

Our contributions to data processing and management in sensor networks are:

- A novel idea, the construction of data cubes as distributed data structures in sensor networks, is proposed. This is a new idea for both sensor networks and distributed systems;
- The proposed technique facilitates the simultaneous retrieval of data aggregates from multiple sets of sensors in the same sensor network;
- The proposed technique allows data aggregate queries to be answered in just a constant number of operations;
- We propose a simple synchronous algorithm for the two proposed distributed data cubes, so that they can be constructed, updated, and queried consistently.

1.5 Thesis Organization

In Chapter 1, we have introduced the background of sensor networks. Since sensor networks are new to many people, we have summarized the properties, capabilities, architectures, and configurations of sensor nodes and sensor networks. In addition, we have stated our research objectives and our contributions.

In Chapter 2, we will present a detailed survey on those research topics related to our research work including data aggregation in sensor networks, data cube, and concurrency control for distributed systems.

In Chapter 3, the two proposed distributed data cubes will be introduced. They are distributed prefix sum data cube and distributed local prefix sum data cube. We will show the construction and querying algorithms for them, and discuss their performances by studying the simulation results. Notice that we will focus on the SUM and AVG (i.e. the sum and the average) aggregation operators in this thesis.

In Chapter 4, we will study data consistency in sensor network systems. We will also propose a synchronous algorithm for the proposed distributed data cubes so that the proposed distributed data cubes can be constructed, updated, and queried consistently

Finally, the conclusion will be drawn in Chapter 5. We will also point out some possible future work based on the technique proposed in this thesis.

Chapter 2

Literature Review

Our work presented in this paper is related to data cube and data aggregation. Data cube is an important technique in traditional database management systems. Therefore, it attracts much attention from researchers [9,23–27]. On the other hand, data aggregation is one of the key challenges in sensor networks. Therefore some researchers have focused on this area and proposed several aggregation methodologies. We are going to introduce the previous work in these two areas. On the other hand, the distributed data cubes proposed in this thesis have to be constructed in sensor networks with grid-like topologies. We will summarize the GAF algorithm proposed by Xu et. al. [29], which is able to transform any sensor network into a grid virtually. Finally, since we will discuss data consistency and concurrency control in sensor networks, hence we will outline the work for concurrency control in distributed systems.

2.1 Data Cube

Data cube [9,23–27] is well studied and widely used in traditional database systems, especially multidimensional database management systems (MDDBMS) [30–32]. MDDBMS is an application of On-Line Analytical Processing (OLAP) [33] which allows fast analysis of aggregate databases storing a huge amount of data.

In MDDBMS, data cube is constructed to maintain multi-dimensional data aggregates. It acts as a data warehouse so that when an aggregate is needed, it can be checked out with relatively fewer operations and calculations. A data cube can be described by two kinds of attributes: a measure attribute and some functional attributes. The measure attribute holds the value of interest, while the functional attributes comprise the dimensions of the multi-dimensional cube. Functional attributes can be treated as the ID for specifying the corresponding measure attribute.

In order to meet different application requirements, several data cube construction algorithms have been proposed. The one that is closely related to this paper is prefix sum (PS) data cube presented in [23] by C. T. Ho et al. PS data cube is the most fundamental data cube, and querying a PS data cube takes only constant time. Relative prefix sum (RPS) [24] and space-efficient relative prefix sum (SRPS) [25] data cubes are similar to PS data cube. They incur lower update costs but greater query costs. When a data cube is partitioned into several parts and a PS data cube is maintained in each of them, then it is a local prefix sum (LPS) data cube [26, 28]. Besides these, there are also data cubes specially designed for dynamic environment, such as dynamic data cube (DDC) [27]. Space-efficient dynamic data cube (SDDC) [25] is an improvement on DDC and it requires less storage space.

2.2 Data Aggregation in Sensor Networks

Data aggregation in sensor networks is one of the most important problems in sensor networks. To the best of our knowledge, three categories of solutions have been proposed. They are hierarchical data aggregation [10, 12], gossip-based aggregation [13], and a mixture of the two [14, 15].

2.2.1 Hierarchical Data Aggregation

Hierarchical data aggregation [10, 12] returns data aggregates by maintaining distributed hierarchical structures in sensor networks, in particular tree-like structures. With these hierarchical structures, data aggregates can be obtained by passing data from leave nodes to the root. At each intermediate node along the path to the root, the aggregate operation may be applied on the data received from the child nodes to form a single aggregate. Then the aggregate is passed to the parent. Since data aggregation is done partially along the path to the root, the network traffic and the workload of the base station are reduced. In this data aggregation model, an update on the reading of any sensor node is passed to the predecessors only. After a certain period of time, the update will eventually reach the root.

2.2.2 Gossip-based Aggregation

Gossip-based aggregation [13] is an interesting aggregation technique. Under this aggregation model, distributed data structures are not used. Instead, it relies on gossip message exchange. When a data aggregate is to be calculated, every node only needs to send data to a randomly chosen neighbor. After a certain period of time and with a high probability, the data aggregate can be calculated. This kind of algorithms can deal with rapid changes in network topology.

2.2.3 Hierarchical Gossip Aggregation

In order to collect data aggregates in an easier way, gossip-based data aggregation techniques with data structures being maintained are proposed [14, 15]. These techniques work well in large scale sensor networks. However, leader election and maintenance algorithms are needed [13]. This introduces overheads on the efficiency and the complexity of the data aggregation process.

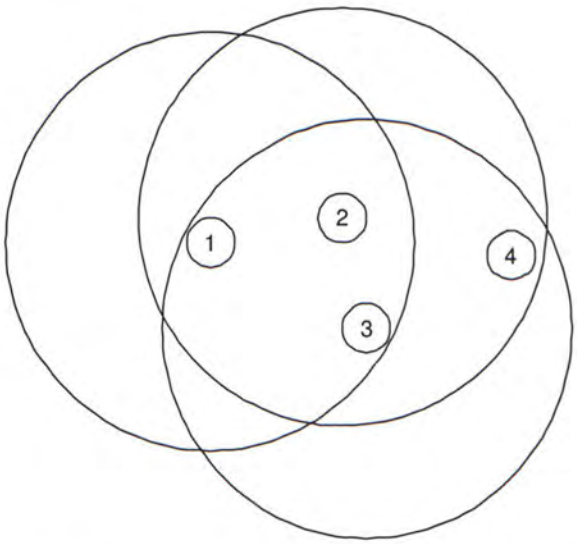


Figure 2.1: Sensor redundancy in sensor network

2.3 GAF Algorithm

The geographical adaptive fidelity (GAF) algorithm [29] was proposed for energy conservation and load balancing in sensor networks, so that the lifetime of any sensor network can be extended. The underlying principle of the GAF algorithm is to switch off redundant sensors so that energy can be saved without destroying the connectivity and functionality of the sensor network. This algorithm is proposed based on the observation that the energy dissipated by redundant and idle sensors cannot be ignored [34, 35]. As an example, in Fig.2.1 sensor 1 needs to communicate with sensor 4 with the help of either sensor 2 or sensor 3. If both sensor 2 and sensor 3 are active, then any of them may stay idle for a long period of time. In order to save energy, we can switch off sensor 2 in order to preserve energy and extend network lifetime

Under the GAF algorithm, a sensor network is partitioned virtually into grids such that all sensors in a grid can communicate directly with the other sensors in the neighboring grids. For the partition to be performed every sensor needs to know its location and the pre-determined grid width r, and r is related to the sensor radio range R. Let us illustrate the idea using Fig.2.2.

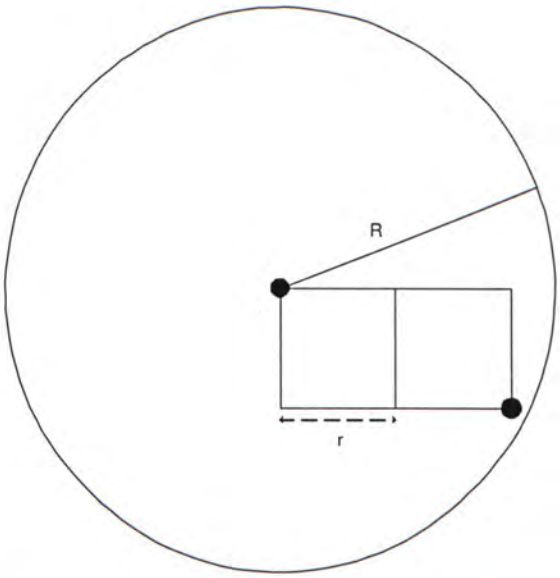


Figure 2.2: Grid width r and sensor radio range R

In the figure, the two sensors are the farthest sensors of the two adjacent grids and R must be large enough for the two sensors to be able to communicate directly. Therefore, r and R are related by:

$$r^2 + (2r)^2 \leq R^2 \Rightarrow r \leq \frac{R}{\sqrt{5}} \tag{2.1}$$

There may be more than one sensor in a grid, and at any time we only allow the sensor with the highest rank or privilege to remain active for a given period of time. For the other redundant sensors, they simply sleep. After some time another selection is performed for the active sensor, the previously active sensor either remains active, or let another sensor to be active if it nearly runs out of energy or another sensor with a higher rank exists in the grid.

The GAF algorithm is independent of the underlying routing protocol, and one of its variations called GAF-ma is adaptable to sensor mobility. Hence by taking the GAF-ma algorithm as the underlying layer, we can transform a mobile sensor network into a grid virtually and hence our proposed techniques can be applied. At the same time, we can enjoy the other benefits, like energy

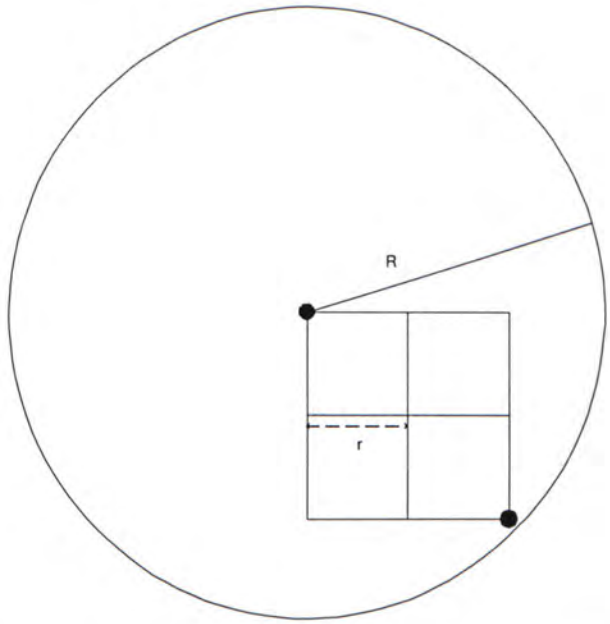


Figure 2.3: Grid width r and sensor radio range R for our proposed techniques

conservation and load balancing, provided by GAF-ma.

However, in order to apply our proposed techniques using GAF-ma, we need to modify the rule to determine the grid width r. It is because for our proposed techniques we assume that the sensor in any grid is able to communicate with the sensors in all the eight neighboring grids, but not just the left, right, upper, and lower neighboring grids.

Let us refer to Fig.2.3, and the new relation between r and R is:

$$(2r)^2 + (2r)^2 \leq R^2 \Rightarrow r \leq \frac{R}{\sqrt{8}} \tag{2.2}$$

Recall that the radio range of the Crossbow sensor nodes we own lies between $30m$ to $100m$, i.e. $30m \leq R \leq 100m$. As a result, when our sensor nodes are used the grid width $r \leq \frac{30}{\sqrt{8}} = 10.61m$.

Finally, readers are recommended to read [29] for more thorough understanding of the GAF algorithm.

2.4 Concurrency Control

Serializability of concurrent execution of read-only and read-write transactions is an important topic in traditional database systems and distributed systems. In the literature many different algorithms for concurrency control of database and distributed systems have been proposed, and they take either the two-phase locking and the timestamp ordering approaches. In this section, we will give an overview on these two approaches for concurrency control in distributed systems.

2.4.1 Two-phase Locking

Under two-phase locking concurrency control, locks are required a priori when a transaction wants to access a data object. There are two kinds of locks, namely *readlock* and *writelock*. When a transaction wants to read a data object, it must first acquire the readlock on that data object. If no transaction is holding the writelock on the data object, the request can be granted and the transaction can read that data object. A data object may grant several readlocks to different transactions at the same time, since a read-only operation does not change the value of the data and hence it does not conflict with other read-only operations. On the other hand, each data object is only granted with one writelock. If a transaction needs to write or update a data object, it has to obtain the writelock on the data object. If no readlocks on that data object are granted, and no transaction is holding the writelock on the data object, then the request of the writelock can be granted. When the read or write operations are completed, the locks must be released. Once a transaction releases a lock, it can no longer make a request for another lock. That is why it is called two-phase locking protocol, as a transaction has to experience the growing phase when lock requests are made, and the shrinking phase when locks are released.

If a transaction wants to acquire either a readlock or a writelock of a data object but it is not available, then the transaction may be forced to wait for the lock [36]. Consider the case where transaction T_1 needs to wait for the lock of transaction T_2, transaction T_2 needs to wait for the lock of transaction T_3, and transaction T_3 needs to wait for the lock of transaction T_1. In this case, a cyclic waiting relation exists. This situation is called *deadlock*, such that all transactions involved in the deadlock wait infinitely unless some of them are aborted. In order to prevent deadlock, deadlock prevention scheme can be applied to avoid deadlock from happening. However, since deadlock may not happen so frequently in reality and deadlock prevention consumes much resources, hence another technique called deadlock detection is commonly adopted. Deadlock detection is a technique aiming at detecting, but not preventing, the existence of deadlock in a system. If deadlock is detected, one of the involved transactions is forced to abort so that other transactions can resume.

2.4.2 Timestamp Ordering

Timestamp ordering based concurrency control protocol is another kind of concurrency control protocol. When a transaction T is issued, it is assigned a unique numerical timestamp $TS(T)$ at startup. On the other hand, each data object O maintains a read timestamp $RTS(O)$ and a write timestamp $WTS(O)$. The idea of concurrency control by timestamp ordering is to select a total order, called *serialization order*, for transaction execution such that transactions are forced to execute in that order [36]. It is proven that if the execution of transactions follows a serialization order, the execution of transactions is serializable [37, 38].

If a transaction T with timestamp $TS(T)$ is issued, it is handled as follows: when T wants to read data object O, $TS(T)$ and $WTS(O)$ will be compared. If $TS(T) \leq WTS(O)$, T is aborted and restarts with a larger timestamp. Otherwise T is allowed to read the data and $RTS(O) = \max(RTS(O), TS(T))$. If T wants to write to data object O, $TS(T)$, $RTS(O)$ and $WTS(O)$ are compared. If $TS(T) \leq WTS(O)$, T is ignored according to the Thomas Write Rule [39], because a write operation of another transaction T' with $TS(T') > TS(T)$ has already been executed and hence T is out-of-date. If $TS(T) \leq RTS(O)$, T conflicts with a read operation already applied on O. Hence T is aborted and it needs to restart with a larger timestamp. If $TS(T) > WTS(O)$ and $TS(T) > RTS(O)$, T is allowed to write on O and $WTS(O) = TS(T)$.

Chapter 3

Building Distributed Data Cubes in Sensor Network

The retrieval of consistent data in sensor networks is a broad topic, and the efficient retrieval of data aggregates is our main research focus. We put our emphasis on the retrieval of data aggregates in sensor networks because, as shown by other researchers, it is an important problem [40–42]. Furthermore, data aggregates are usually more useful than individual sensor values [9–11]. Therefore instead of individual sensor value, data aggregates are more usually wanted.

Since data aggregation is the core of data processing and management in sensor networks, it has attracted much attention. As we have introduced in Chapter 2, in the literature three classes of aggregation techniques have been proposed. They are hierarchical data aggregation [10, 12], gossip-based aggregation [13], and a mixture of the two [14, 15]. These existing data aggregation techniques are able to return the data aggregates of a single set of sensors. However, they are not feasible to be used when we want to retrieve data aggregates from more than one set of sensors. Consider a sensor network with N sensor nodes. If we apply the existing aggregation techniques to retrieve the data aggregates from all of the 2^N possible sets of sensors in the sensor network, in worst case we need to build an aggregation tree or gossip group in

each of the sets. As a result, we will have up to 2^N trees or gossip groups in the sensor network.

From this example we can see that existing techniques are too expensive to be used for solving the above problem. However, we do not mean that existing techniques are not good at all. Actually, sensor networks are application specific [43]. Hence for applications requiring the data aggregates of only one set of sensors, existing techniques are feasible. On the other hand, new algorithms are needed for applications that require the data aggregates of multiple sets of sensors.

In this chapter, we are going to propose a new class of techniques for the retrieval of data aggregates in sensor networks. The techniques are capable of solving the problem described above, so that the fast and simultaneous retrieval of data aggregates from multiple regions in a sensor network can be achieved. Our idea is inspired by the use of data cubes [9, 23–27] in the traditional database systems, and we propose to construct distributed prefix sum data cube and distributed local prefix sum data cube in sensor networks. Our idea was previously published in [44].

3.1 Aggregation Operators

In different research papers on data aggregation in sensor networks, one may notice that data aggregates may refer to different data aggregation operations. It is because different researchers often have different definitions for their own sets of data aggregation operators subjected to different assumptions and requirements. For example, in [45] data aggregates are defined to be the sum, the average, the minima, or the maxima of a set of sensor values. In [13], the aggregation operators listed above are considered together with other operations such as random sample and quantile. In this thesis, we consider two fundamental aggregation operators: aggregate sum and aggregate average.

The aggregate sum $Sum(\mathbf{X})$ of a set of sensor values $\mathbf{X}$, $\mathbf{X} = \{x_i\}$ where $i \geq 1$, is the total sum of all x_i:

$$Sum(\mathbf{X}) = \sum x_i \tag{3.1}$$

Similarly the aggregate average $Avg(\mathbf{X})$ of a set of sensor values $\mathbf{X}$, $\mathbf{X} = \{x_i\}$ where $i \geq 1$, is the arithmetic mean of all x_i:

$$Avg(\mathbf{X}) = \frac{\sum x_i}{i} \tag{3.2}$$

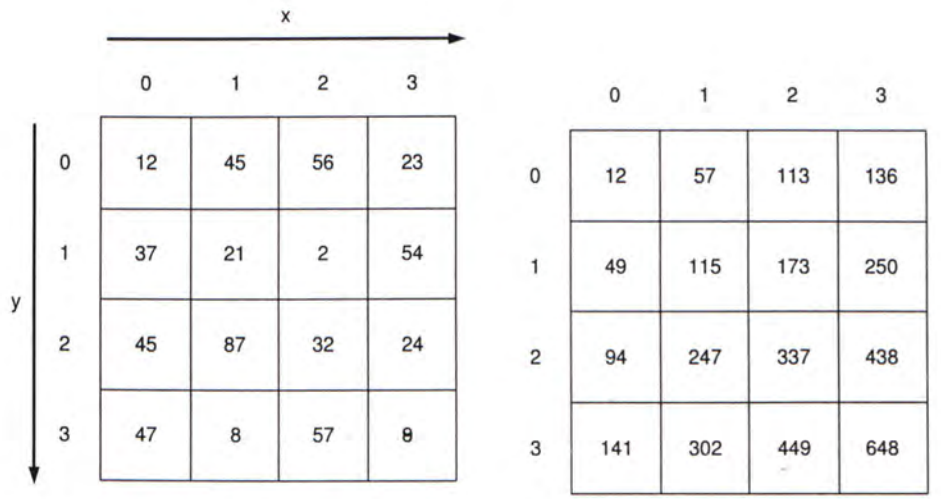
These two operators are selected and handled because data cubes are basically designed to help answer aggregate sum queries. And as will be shown in the later part of this thesis, the aggregate average of a set of sensor values can be answered once we get their aggregate sum.

3.2 Distributed Prefix (PS) Sum Data Cube

Data cubes are commonly used in traditional database systems for the fast retrieval of data aggregates. We are inspired by the use of data cubes in the traditional database systems, and prefix sum data cube is the most fundamental one. To help understand distributed prefix sum data cube, we will first introduce the working principle of prefix sum data cube. Then we will present the construction and querying algorithms, and the simulation results of the proposed distributed prefix sum data cube.

3.2.1 Prefix Sum (PS) Data Cube

Consider a 2-D grid in which every cell holds a numerical value, and let $v(x, y)$ be the numerical value kept by the cell at column x and row y (i.e. (x, y)). A naive approach to obtain the aggregate sum of the values in a particular



(a) (b)

Figure 3.1: (a) A grid-like area and the readings of the cells; (b) The PS data cube of that area

region is to query all the cells in that region and compute the aggregate sum after all the replies have arrived.

Prefix sum (PS) data cube [23] stores pre-computed data, called *prefix sums*, so that the aggregate sum of any region can be calculated quickly with the pre-computed data. A PS data cube for the grid in Fig.3.1(a) is shown in Fig.3.1(b). Each cell in Fig.3.1(b) keeps a prefix sum, which is equal to the sum of the values held by those cells locating at the upper left corner of the corresponding cell in the original grid. For example, the cell at $(1, 2)$ in Fig.3.1(b) keeps the sum of $v(0, 0)$, $v(1, 0)$, $v(0, 1)$, $v(1, 1)$, $v(0, 2)$, and $v(1, 2)$ in Fig.3.1(a), i.e. $12 + 45 + 37 + 21 + 45 + 87 = 247$.

Here are the definitions of prefix sum and prefix sum data cube, which were introduced in [23]:

Definition 1 In a 2-D grid in which each cell i locating at column x_i and row y_i keeps a value $v(x_i, y_i)$, the prefix sum $s(x_i, y_i)$ stored in cell i is $\sum_{m=0}^{x_i} \sum_{n=0}^{y_i} v(m, n)$.

□

Definition 2 A prefix sum data cube is a 2-D grid of cells in which each cell maintains a prefix sum.

□

3.2.2 Notations

Any rectangular region can be identified by two cells e and f, located at the upper left corner and bottom right corner of that region respectively. Usually, e and f are called the *anchor* and the *endpoint* of that region [25], and the expression $e : f$ is the *region* specified by the anchor e and endpoint f [25].

Definition 3 $e : f$ is the region specified by the anchor e and endpoint f.

□

If the anchor e and the endpoint f of the region $e : f$ are located at (x_e, y_e) and (x_f, y_f) respectively, then any cell c at (x_c, y_c) is in $e : f$ if and only if $x_e \leq x_c \leq x_f$ and $y_e \leq y_c \leq y_f$.

Definition 4 A cell c at (x_c, y_c) is in $e : f$ if and only if $x_e \leq x_c \leq x_f$ and $y_e \leq y_c \leq y_f$.

□

For example, assume that e and f are the cells with coordinates $(0, 0)$ and $(2, 1)$ respectively, then $e : f$ is the region containing the cells at $(0, 0)$, $(1, 0)$, $(2, 0)$, $(1, 0)$, $(1, 1)$, and $(2, 1)$.

Since every cell i holds a numerical value $v(x_i, y_i)$, so we can define the aggregate sum $Sum(e : f)$ of all the cell values in any region $e : f$ as:

Definition 5 The aggregate sum of all the cell values in a region $e : f$ is $Sum(e : f)$ and it is given by:

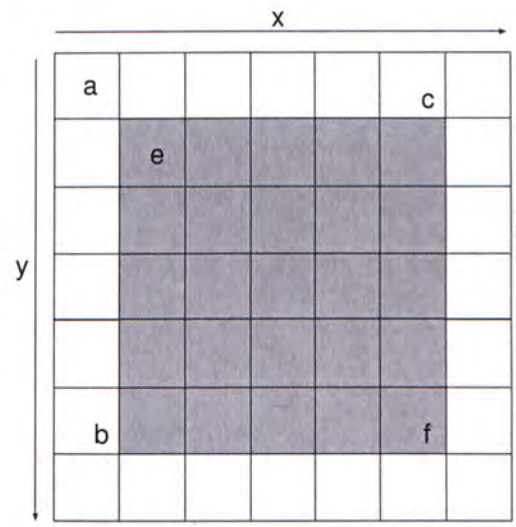


Figure 3.2: To calculate the aggregate sum of a region

$$Sum(e:f) = \sum_{m=x_e}^{x_f} \sum_{n=y_e}^{y_f} v(m,n) \tag{3.3}$$

□

As stated before, the cell value and the prefix sum of any cell i locating at (x_i, y_i) are $v(x_i, y_i)$ and $s(x_i, y_i)$ respectively. For simplicity the symbols $v(i)$ and $s(i)$ will also be used interchangeably in the rest of this thesis so that $v(x_i, y_i) = v(i)$ and $s(x_i, y_i) = s(i)$.

3.2.3 Querying a PS Data Cube

Suppose we have a PS data cube and if we want to retrieve the aggregate sum $Sum(e:f)$ from a rectangular region $e:f$, we only need to issue queries to at most four cells in the PS data cube. Then based on the returned values, $Sum(e:f)$ can be computed. In Fig.3.2, each cell in the grid stores both the source data value and the prefix sum so that the grid acts as the source data grid as well as the PS data cube at the same time. The aggregate sum

$Sum(e : f)$ of the shaded region can be calculated using only four values in the PS data cube by the following formula:

$$Sum(e : f) = s(f) - s(b) - s(c) + s(a) \tag{3.4}$$

Equation 3.4 can be rewritten in terms of merely the anchor e and the endpoint f of the region:

$$\begin{aligned} Sum(e : f) = {} & s(x_f, y_f) - s(x_e - 1, y_f) \\ & - s(x_f, y_e - 1) \\ & + s(x_e - 1, y_e - 1) \end{aligned} \tag{3.5}$$

The general proof of Equation 3.5 for an N-dimensional PS data cube is in [23]. However, for the sake of completeness we would like to show another way to prove it for a 2-D grid of data.

Lemma 1

$$Sum(e : f) = s(x_f, y_f) - s(x_e - 1, y_f) - s(x_f, y_e - 1) + s(x_e - 1, y_e - 1)$$

Proof of Lemma 1

$Sum(e : f)$

$= \sum_{m=x_e}^{x_f} \sum_{n=y_e}^{y_f} v(m, n)$

$= \sum_{m=x_e}^{x_f} \sum_{n=0}^{y_f} v(m, n) - \sum_{m=x_e}^{x_f} \sum_{n=0}^{y_e-1} v(m, n)$

$= \sum_{m=x_e}^{x_f} \sum_{n=0}^{y_f} v(m, n) - \sum_{m=0}^{x_f} \sum_{n=0}^{y_e-1} v(m, n) + \sum_{m=0}^{x_e-1} \sum_{n=0}^{y_e-1} v(m, n)$

$= \sum_{m=0}^{x_f} \sum_{n=0}^{y_f} v(m, n) - \sum_{m=0}^{x_e-1} \sum_{n=0}^{y_f} v(m, n) - \sum_{m=0}^{x_f} \sum_{n=0}^{y_e-1} v(m, n) + \sum_{m=0}^{x_e-1} \sum_{n=0}^{y_e-1} v(m, n)$

$= s(x_f, y_f) - s(x_e - 1, y_f) - s(x_f, y_e - 1) + s(x_e - 1, y_e - 1)$ (By Definition 1)

□

For instance, the aggregate sum of the region $(1, 1) : (3, 3)$ in the source grid in Fig.3.1(a) is $s(3, 3) - s(0, 3) - s(3, 0) + s(0, 0) = 383$, where $s(x, y)$ is the prefix sum of the cell at (x, y) of the PS data cube in Fig.3.1(b).

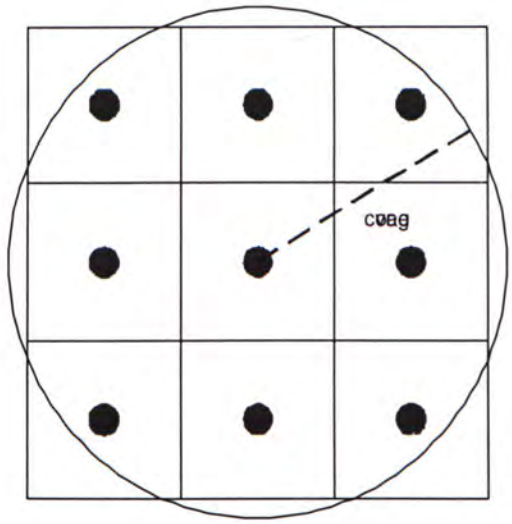


Figure 3.3: A sensor can communicate with its immediate neighbors

3.2.4 Building Distributed PS Data Cube

The working principle of PS data cube have been introduced, and we have also shown how the aggregate sum of a rectangular region in a grid can be computed using a PS data cube. Now we will present the proposed technique for building distributed data cube in sensor networks. It is supposed that the sensor network, where a distributed PS data cube is to be built, has a grid-like topology such that each cell contains a sensor (readers may refer to Section 2.3 for the virtual transformation of any sensor network to a grid). Besides that, each sensor stores its sensor value, prefix sum, and prefix average altogether in its memory. It is also assumed that each sensor can communicate with its immediate neighbors (Fig.3.3) and broadcast messages to them. With our proposed algorithm, a distributed PS data cube can be built autonomously for the fast and simultaneous retrieval of data aggregates.

Since every sensor in a PS distributed data cube stores a prefix sum and a prefix average, and these two prefix values have different properties, therefore we need to handle the values differently. We will first describe how prefix sum can be maintained, followed by an explanation on prefix average.

Maintaining Prefix Sum

Consider the sensors in Fig.3.4. Assume that the sensor values stored in sensors a, b, c, and d are $v(a)$, $v(b)$, $v(c)$, and $v(d)$ respectively, and their prefix sums are $s(a)$, $s(b)$, $s(c)$, and $s(d)$ respectively. $s(d)$ includes the sensor values of all sensors at the upper left corner of sensor d. Therefore the simplest way for sensor d to compute $s(d)$ is as follows:

1. sends queries to these sensors at the upper left corner of sensor d;
2. receives replies from these sensors;
3. finds out the sum of these sensor values;
4. adds $v(d)$ to the sum of sensor values.

The procedures shown above are viable. However, a sensor network may contain more than ten thousand sensors and hence it is not a feasible solution.

With our analysis, we found that $s(d)$ can be computed easily using the values stored in the neighbors of sensor d, such that only $v(d)$, $s(a)$, $s(b)$, and $s(c)$ are required. To do so, Lemma 2 is needed, and Lemma 2 is based on the inclusion-exclusion principle.

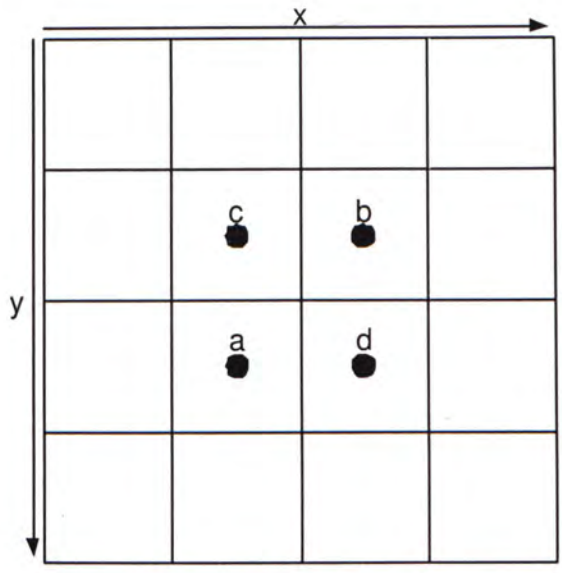


Figure 3.4: Sensors a, b, c, and d

Lemma 2 $s(x_d, y_d) = v(x_d, y_d) + s(x_d - 1, y_d) + s(x_d, y_d - 1) - s(x_d - 1, y_d - 1)$ for any cell d located at (x_d, y_d).

Proof of Lemma 2

$s(x_d, y_d)$

$= \sum_{m=0}^{x_d} \sum_{n=0}^{y_d} v(m, n)$

$= \sum_{m=0}^{x_d-1} \sum_{n=0}^{y_d} v(m, n) + \sum_{n=0}^{y_d} v(x_d, n)$

$= v(x_d, y_d) + \sum_{m=0}^{x_d-1} \sum_{n=0}^{y_d} v(m, n) + \sum_{n=0}^{y_d-1} v(x_d, n)$

$= v(x_d, y_d) + \sum_{m=0}^{x_d-1} \sum_{n=0}^{y_d} v(m, n) + \sum_{m=0}^{x_d} \sum_{n=0}^{y_d-1} v(m, n) - \sum_{m=0}^{x_d-1} \sum_{n=0}^{y_d-1} v(m, n)$

$= v(x_d, y_d) + s(x_d - 1, y_d) + s(x_d, y_d - 1) - s(x_d - 1, y_d - 1)$ (By Definition 1)

□

In general, for any sensor d with $x_d, y_d \geq 0$, $s(x_d, y_d)$ can be obtained by Lemma 2. In the proof it is assumed that $s(x_d, y_d) = 0$ when $x_d < 0$ or $y_d < 0$.

Algorithm 1 Maintain Prefix Sum

terminate = FALSE
$l(i)$ = empty, $u(i)$ = empty, $d(i)$ = empty

repeat
  if $s(j)$ from sensor j is received **then**
    if j is on the left of i **then**
      $l(i) = s(j)$
    end if
    if j is on top of i **then**
      $u(i) = s(j)$
    end if
    if j is on the upper left of i **then**
      $d(i) = s(j)$
    end if
    if $l(i)$, $u(i)$ and $d(i)$ are not empty **then**
      $s(i) = v(i) + l(i) + u(i) - d(i)$
      terminate = TRUE
    end if
  end if
until terminate = TRUE

i broadcasts $s(i)$ to j_1 where j_1 is on the right of i
i broadcasts $s(i)$ to j_2 where j_2 is at the bottom of i
i broadcasts $s(i)$ to j_3 where j_3 is at the bottom right of i

The distributed PS data cube construction algorithm that we are going to introduce now is based on Lemma 2: consider a grid-like 2-D sensor network, in which each node i maintains a prefix sum $s(i)$, its own sensor value $v(i)$, and three variables, namely $u(i)$, $l(i)$ and $d(i)$, where $u(i)$, $l(i)$ and $d(i)$ store the prefix sums received from the upper, left, and the upper left neighbors (i.e. sensors b, a and c respectively for sensor d in Fig.3.4). At time $t = 0$, the sensor node at $(0, 0)$ initializes $s(0, 0)$, and $s(0, 0) = v(0, 0)$. Then it broadcasts $s(0, 0)$ to its neighbors. Once a sensor receives the prefix sums from its upper, left and upper left neighbors, it will update the variables $u(i)$, $l(i)$, and $d(i)$ respectively according to Algorithm 1. After that the sensor can compute for its own prefix sum, and then broadcast the prefix sum to its neighbors. As Algorithm 1 is executed repeatedly, the prefix sums of all the sensors can be maintained.

Maintaining Prefix Average

The prefix average of any sensor i at (x_i, y_i) is the average of all the sensor values of the sensors locating at the upper left corner of sensor i. In other words, let $m(i)$ be the prefix average of sensor i and $pxPop(i)$ be the prefix sensor population (i.e. the number of sensors locating at the upper left corner of i). Then $m(i)$ can be computed using $s(i)$ and $pxPop(i)$ of sensor i:

$$m(i) = \frac{s(i)}{pxPop(i)} \tag{3.6}$$

where

$$pxPop(i) = (x_i + 1) \times (y_i + 1) \tag{3.7}$$

The prefix sum $s(i)$ of i can be obtained using Algorithm 1. On the other hand, sensor i already knows $pxPop(i)$ because the sensor network is a grid and $pxPop(i)$ can be derived from x_i and y_i by Equation 3.7. As a result,

with Algorithm 1 the prefix averages of all the sensors can be calculated at the same time when the prefix sums are maintained. That means Algorithm 1 is capable of maintaining the prefix averages, as well as the prefix sums, of sensors without significant overheads.

3.2.5 Time Bounds

In order to know the time complexities for the construction and update of a distributed PS data cube in a sensor network, theoretical analysis was done.

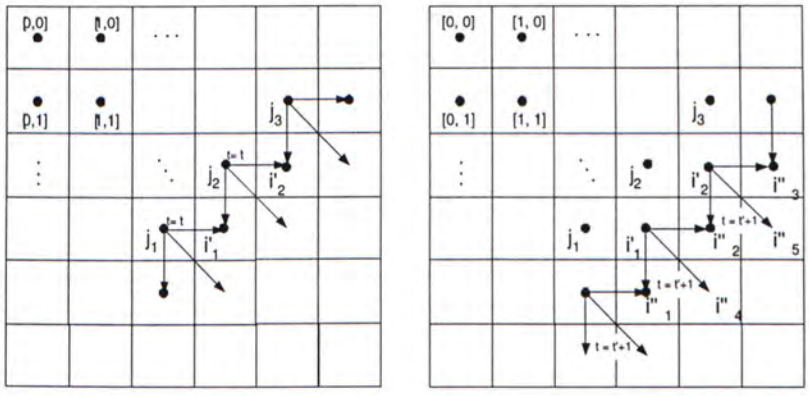
Construction Time

Assume that the time bound for a sensor to calculate and broadcast its prefix sum and prefix average, and for the broadcasted prefix values to be received by all of its immediate neighbors is 1. We observed that the number of sensors with their prefix values computed was closely related to the number of time units that the construction algorithm had been run.

Lemma 3 In a distributed PS data cube, assume that the time bound for a sensor to calculate and broadcast its prefix sum and prefix average, and for the broadcasted prefix values to be received by all its neighbors is 1. At any time t, the prefix values of all the sensor nodes locating at (x, y), such that $x + y = t$, will be ready.

Proof of Lemma 3 We will prove the Lemma by showing that the prefix sums of all the sensor nodes locating at (x, y), such that $x + y = t$, will be ready at time unit t. Since prefix averages can be deduced from prefix sums, hence the statement will be valid for prefix averages if it holds for prefix sums.

At $t = 0$, $s(0, 0)$ is already ready, and it is simply equivalent to $v(0, 0)$. sensor $(0, 0)$ then sends $s(0, 0)$ to sensors $(0, 1)$, $(1, 0)$ and $(1, 1)$. At $t = 1$, sensors $(0, 1)$ and $(1, 0)$ update $s(0, 1)$ and $s(1, 0)$ correspondingly. Since sensor



(a) (b)

Figure 3.5: Prefix sums are being calculated in the distributed data cube

$(1,1)$ has not received $s(0,1)$ and $s(1,0)$, so it cannot compute $s(1,1)$. The statement is true at time $t = 1$ because sensors $(0,1)$ and $(1,0)$ obtained their prefix sums at time unit 1.

Assume the statement also holds at time $t = t'$ so that node i'_1 has received prefix sums $s(j_1)$ and $s(j_2)$ from sensors j_1 and j_2 (see Fig.3.5(a)) such that $x_{i'_1} = x_{j_1} + 1$ and $y_{i'_1} = y_{j_1}$, $x_{i'_1} = x_{j_2}$ and $y_{i'_1} = y_{j_2} + 1$. i'_1 then obtains $s(i'_1)$ by $s(i'_1) = v(i'_1) + s(j_1) + s(j_2) - s(k_1)$ where $x_{k_1} = x_{i'_1} - 1$ and $y_{k_1} = y_{i'_1} - 1$.

At the same iteration, node i'_2 has also received prefix sums $s(j_2)$ and $s(j_3)$ from sensors j_2 and j_3 such that $x_{i'_2} = x_{j_2} + 1$ and $y_{i'_2} = y_{j_2}$, $x_{i'_2} = x_{j_3}$ and $y_{i'_2} = y_{j_3} + 1$. i'_2 obtains $s(i'_2)$ by $s(i'_2) = v(i_2) + s(j_2) + s(j_3) - s(k_2)$ where $x_{k_2} = x_{i'_2} - 1$ and $y_{k_2} = y_{i'_2} - 1$. Notice that $x_{i'_1} + y_{i'_1} = x_{i'_2} + y_{i'_2} = t'$. After that i'_1 sends its prefix sum $s(i'_1)$ to i''_1, i''_2 and i''_4, i'_2 sends $s(i'_2)$ to i''_2 , i''_3 and i''_5 (see Fig.3.5(b)).

At time $t' + 1$, i''_2 can compute $s(i''_2)$ by $s(i''_2) = v(i''_2) + s(i'_1) + s(i'_2) - s(j_2)$. Notice that $x_{i''_2} + y_{i''_2} = x_{i'_1} + y_{i'_1} + 1 = t' + 1$. At the same iteration, all other sensors k where $x_k + y_k = x_{i''_2} + y_{i''_2}$ can also compute for s_k. As a result, the statement holds at time $t = t' + 1$.

Therefore, Lemma 3 is true by the principle of mathematical induction.

□

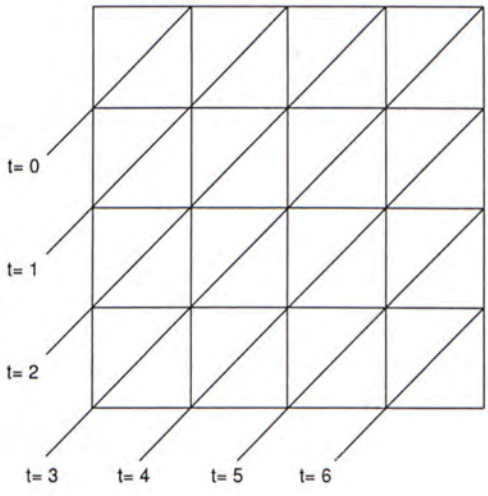


Figure 3.6: Sensors with their prefix values computed, and the time when the prefix values were computed

From the result of Lemma 3, we know that a distributed PS data cube is constructed with a pattern as shown in Fig.3.6. It is like flowing water from the upper left corner and the water will diffuse gradually to the bottom right corner. Furthermore, from Lemma 3 we can deduce that the construction time of a distributed PS data cube is bounded by a sharp time bound, which depends on the size of the distributed PS data cube (i.e. the length h and width k).

Lemma 4 In a sensor network with length $= h$ cells and width $= k$ cells, a distributed PS data cube can be completely constructed in $h + k - 2$ units of time.

Proof of Lemma 4 From Lemma 3, at time unit t the prefix values of every sensor i at (x_i, y_i) with $x_i + y_i = t$ will be computed. In any grid-like sensor network, the bottom-rightmost sensor is at $(h-1, k-1)$ (given that the sensor at the top left corner is at $(0, 0)$). For any sensor i in the sensor network, $x_i \leq h - 1$ and $y_i \leq k - 1$. Hence just before $t = h + k - 2$, the sensor at

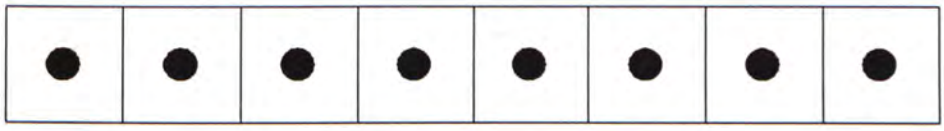


Figure 3.7: A 1-D sensor network

$(h-1, k-1)$ is the only sensor in the sensor network with its prefix values not yet computed, and the prefix values of that sensor will be computed at $t = h + k - 2$. As a result, the whole distributed PS data cube can be built in $h + k - 2$ units of time.

□

From Lemma 4, the time complexity of the proposed distributed PS data cube algorithm can be known: when the distributed PS data cube is a square, i.e. $h \approx k \approx \sqrt{N}$ where N is the number of sensors, the time bound is $O(\sqrt{N})$. In the worst case where the network grid is like a 1-D array (Fig.3.7), the construction time is $N - 1$.

Update Time

Sensors in a sensor network may update their sensor values continuously as required by the deployment scheme, or just when they detect a change in the environment they monitor. Since the sensor value of a sensor may be included in the calculation of the prefix values of some other sensors, hence an update in the sensor value on any sensor should be propagated to the other sensors locating at the bottom right corner of it. The propagation of an updated sensor value can be achieved using Algorithm 1, by assuming the sensor being updated locate at the top left corner (Fig.3.8). By doing so, we can measure the time need for a distributed PS data cube to be updated completely.

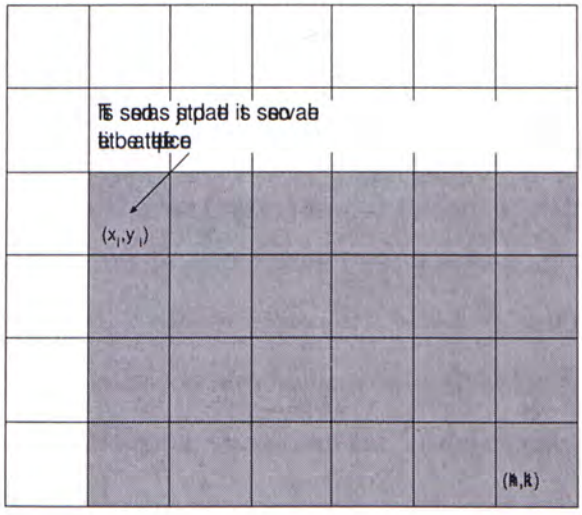


Figure 3.8: A sensor has just updated its sensor value

Lemma 5 In a distributed PS data cube with length $= h$ cells and width $= k$ cells, if sensor i at (x_i, y_i) updates its sensor value $v(i)$, it takes $h+k-x_i-y_i-2$ time for the update to be propagated to all other sensors j at (x_j, y_j), where $x_i \leq x_j \leq h-1$ and $y_i \leq y_j \leq k-1$.

Proof of Lemma 5 The propagation of the update in sensor i to all other sensors j, where $x_i \leq x_j \leq h-1$ and $y_i \leq y_j \leq k-1$, is equivalent to the construction of a new distributed PS prefix sum data cube with i staying at the top left corner (Fig.3.8). Since the sensor at the bottom right corner is at $(h-1, k-1)$, hence we can see that the dimension of the new distributed PS data cube will be $h-1-(x_i-1) = h-x_i$ cells times $k-1-(y_i-1) = k-y_i$ cells.

From Lemma 4, a distributed PS data cube with length $= h-x_i$ cells and width $= k-y_i$ cells can be constructed in $h-x_i+k-y_i-2 = h+k-x_i-y_i-2$ units of time. As a result, it takes $h+k-x_i-y_i-2$ units of time for the update at sensor i to be propagated to all other sensors j, where $x_i \leq x_j \leq h-1$ and $y_i \leq y_j \leq k-1$.

□

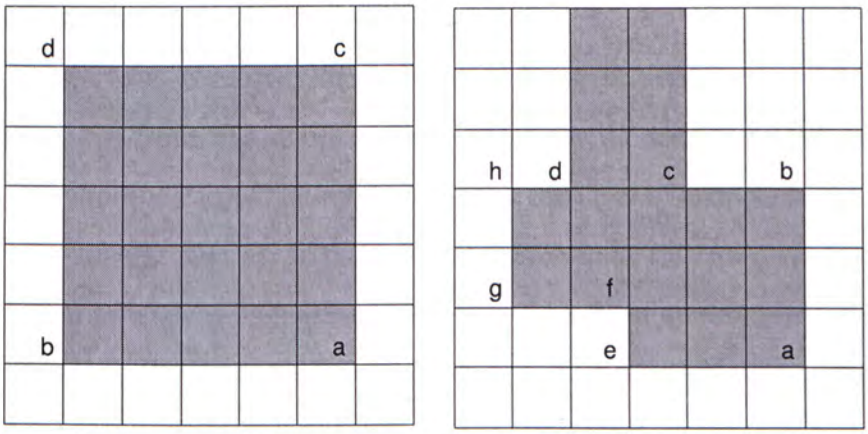
From Lemma 5, we can observe that in a sensor network with $h \times k$ sensors, it takes $h + k - x_i - y_i - 2$ units of time for an update at sensor i to be propagated to all other sensors. In worst case, if the update happens at the sensor at $(0, 0)$, then it takes as much time to update the distributed PS data cube as to construct the whole distributed PS data cube. If the sensor network is like a 1-D array (Fig.3.7), then it takes $O(N)$ time to update the distributed PS data cube. This property is similar to that of the original PS data cube in traditional database systems, such that the update cost of PS data cube is relatively high.

3.2.6 Fast Aggregate Queries on Multiple Regions

When a distributed PS data cube is constructed, it can facilitate the fast and simultaneous retrieval of aggregate sums and aggregate averages from multiple regions in a sensor network, as the prefix values of sensors are pre-computed. An example is shown in Fig.3.9(a). To answer the aggregate sum query on the shaded area in Fig.3.9(a), sensor values of a, b, c and d are needed and the aggregate sum is $s(a) - s(b) - s(c) + s(d)$.

Our proposed technique is different from the existing ones proposed previously (like those proposed in [10, 12–15]) as our technique can handle the fast and simultaneous retrieval of aggregate sums and aggregate averages from several overlapping or non-overlapping regions in the same sensor network using only one distributed data structure.

To illustrate the point, we can investigate Fig.3.10. In the figure, there are three overlapping shaded regions. Using existing techniques, we need to maintain a hierarchical structure in each shaded region or exchanging gossip messages within each region at the same time. Now using our proposed technique we only need to build one distributed data structure, that is a distributed PS data cube, in the sensor network and the data aggregates can be computed



(a) (b)

Figure 3.9: The aggregate sum of the shaded areas = (a) $s(a)-s(b)-s(c)+s(d)$; (b) $s(a) - s(b) + s(c) - s(d) - s(e) + s(f) - s(g) + s(h)$

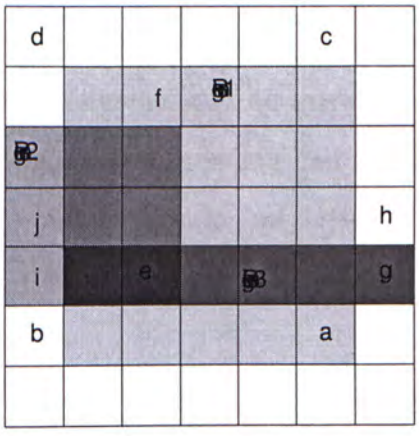


Figure 3.10: Querying the aggregate sums of several regions simultaneously

using the prefix values of sensors a to j. For instance, the following equations show the aggregate sums $Sum(Region1)$, $Sum(Region2)$, and $Sum(Region3)$ of Region 1, Region 2, and Region 3 respectively:

$$Sum(Region1) = s_a - s_b - s_c + s_d$$

$$Sum(Region2) = s_e - s_f$$

$$Sum(Region3) = s_g - s_h - s_i + s_j$$

Up to now the examples we have shown are limited to the retrieval of data aggregates from rectangular areas. However, it does not mean that the proposed technique cannot be applied for the retrieval of data aggregates from areas with irregular shapes. In fact, we can tailor make queries on sensor values in order to answer data aggregate queries on regions of any shapes. Fig.3.9(b) is another example showing how queries can be tailor made to obtain the aggregate sum of a region with arbitrary shape. To compute the aggregate sum of that region with irregular shape, we can retrieve the prefix sums of sensors a to h and the desired aggregate sum is equal to $s(a) - s(b) + s(c) - s(d) - s(e) + s(f) - s(g) + s(h)$.

Now we will explain how the aggregate sum and aggregate average of any region in a grid-like sensor network can be answered using a distributed PS data cube. From our explanation, readers can know how a aggregate sum or aggregate average query can be tailor made to suit an area with irregular shape.

Aggregate Sum Query

The prefix sum $s(i)$ of any sensor i is equal to the aggregate sum of the rectangular region bounded by the sensor at $(0, 0)$ and i. As a result, to obtain the aggregate sum of the region bounded by the sensor at $(0, 0)$ and any sensor i, we only need to know the value of $s(i)$ and it is the aggregate sum of that region.

For other cases, data from at most four sensors are needed to answer an aggregate sum query on any rectangular region. Consider Fig.3.9(a), the aggregate sum of the shaded region is $s(a)$ minus the sum of the sensor values of the unshaded region bounded by sensors b, d, and c. We can subtract $s(b)$ and $s(c)$ from $s(a)$, but the value held by sensor d would be deducted twice. So we add $s(d)$ and hence the aggregate sum of the shaded region is

Algorithm 2 Aggregate Sum Querying Algorithm on a Rectangular region $e : f$

var $l_{max}, l_{min}, r_{max}, r_{min}$
var $Sum(e : f)$; // the aggregate
$l_{max} = \max(x_e, x_f)$
$l_{min} = \min(x_e, x_f)$
$r_{max} = \max(y_e, y_f)$
$r_{min} = \min(y_e, y_f)$

if $l_{min} = 0$ and $r_{min} = 0$ **then**
$Sum(e : f) = s(i)$ where $(x_i, y_i) = (l_{max}, r_{max})$
end if

if $l_{min} = 0$ and $r_{min} \neq 0$ **then**
$Sum(e : f) = s(i) - s(j)$ where $(x_i, y_i) = (l_{max}, r_{max})$ and $(x_j, y_j) = (l_{max}, r_{min} - 1)$
end if
if $l_{min} \neq 0$ and $r_{min} = 0$ **then**
$Sum(e : f) = s(i) - s(j)$ where $(x_i, y_i) = (l_{max}, r_{max})$ and $(x_j, y_j) = (l_{min} - 1, r_{max})$
end if
if $l_{min} \neq 0$ and $r_{min} \neq 0$ **then**
$Sum(e : f) = s(i) - s(j) - s(k) + s(l)$ where
$(x_i, y_i) = (l_{max}, r_{max})$ and
$(x_k, y_k) = (l_{max}, r_{min} - 1)$ and
$(x_j, y_j) = (l_{min} - 1, r_{max})$ and
$(x_l, y_l) = (l_{min} - 1, r_{min} - 1)$
end if

Construction Cost	Update Cost	Query Cost (rectangular regions)
$N-1$	$N-1$	4

Table 3.1: The worst case construction cost, update cost, and query cost of a distributed PS data cube with N sensors

$s(a) - s(b) - s(c) + s(d)$. Notice that, similar to the construction algorithm, the inclusion-exclusion principle is used for querying the aggregate sum of any region in a sensor network. The algorithm for the retrieval of aggregate sum from any rectangular region is shown in Algorithm 2.

To handle an aggregate sum query on a region with irregular shape, we need to know in advance which sensors have to be queried, so that we can get the required prefix sums for the calculation of the aggregate sum based on the inclusion-exclusion principle. In any cases, we can get the aggregate sum of a region with a bounded number of queries, and the value can be easily estimated once we know the number of sensors that we need to query.

The worst case construction cost, update cost, and query cost of distributed PS data cube is summarized in Table 3.1.

Aggregate Average Query

The prefix sum $s(i)$ of sensor i equals the aggregate sum of the rectangular region bounded by the sensor at $(0,0)$ and i. Similarly the prefix average $m(i)$ of any sensor i equals the aggregate average of the rectangular region bounded by the sensor at $(0,0)$ and i. Therefore, in order to obtain the aggregate average of the rectangular region bounded by the sensor at $(0,0)$ and sensor i, we need to query for $m(i)$ and it is the aggregate average of that region.

However, if we want to obtain the aggregate average of a region not bounded by $(0,0)$, the prefix averages of sensors are no longer helpful since it is di cult to calculate the aggregate average of a region using the prefix averages of sensors. Instead, we need to rely on the prefix sums and the prefix sensor populations of the sensors at the corners of that region. Consider Fig.3.9(a)

again. To obtain the aggregate average of the shaded region, we can first retrieve the aggregate sum of that region. Then we can compute the number of sensors in that region using the x,y-coordinates of the anchor and endpoint of that region. After that, the aggregate average of the shaded region can be calculated.

Generally, given any rectangular region $e : f$ bounded by the anchor e and endpoint f, the number of sensors $SenPop(e : f)$ in $e : f$ is given by:

$$SenPop(e : f) = (x_f - x_e + 1) \times (y_f - y_x + 1) \tag{3.8}$$

For example, the number of sensors in the region bounded by $(1, 2)$ and $(5, 4)$ is $(5 - 1 + 1) \times (4 - 2 + 1) = 15$.

After obtaining $Sum(e : f)$ and $SenPop(e : f)$, the aggregate average $Avg(e : f)$ of that region would then be:

$$Avg(e : f) = \frac{Sum(e : f)}{SenPop(e : f)} \tag{3.9}$$

To compute $Sum(e : f)$ and $SenPop(e : f)$ of a rectangular region, we only need to send queries to not more than four sensors. Hence, same as the case of $Sum(e : f)$, the computation of $Avg(e : f)$ can also be done in a constant number of operations.

If we want to know the aggregate average of a region with irregular shape, we would need to get the aggregate sum of that region first. Then we need to obtain the number of sensors in that region, by performing some additions and subtractions. After that the aggregate average can be obtained. Since finding the number of sensors in a region of irregular shape is trivial, we do not discuss it in this thesis.

3.2.7 Simulation Results

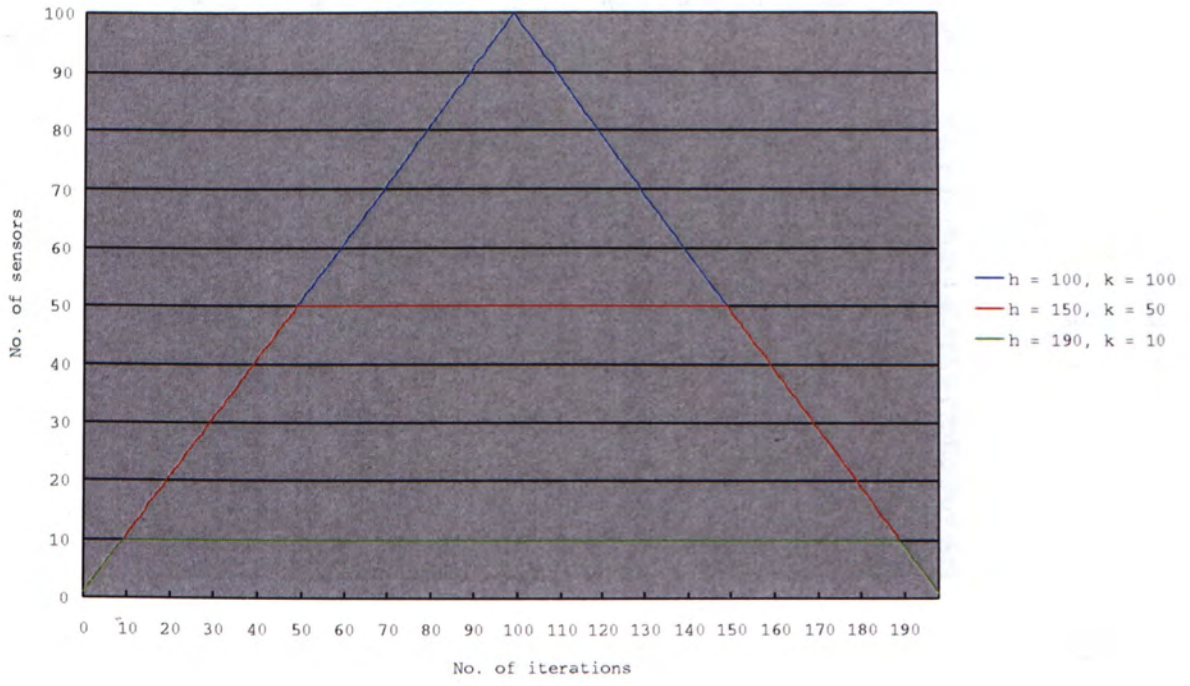
A series of simulations on the proposed technique were performed, and the simulations focused on the time needed to construct distributed PS data cubes, the network traffic incurred, and the scalability of the proposed technique. In this section, we will show and discuss the simulation results.

Data Cube Construction Speed

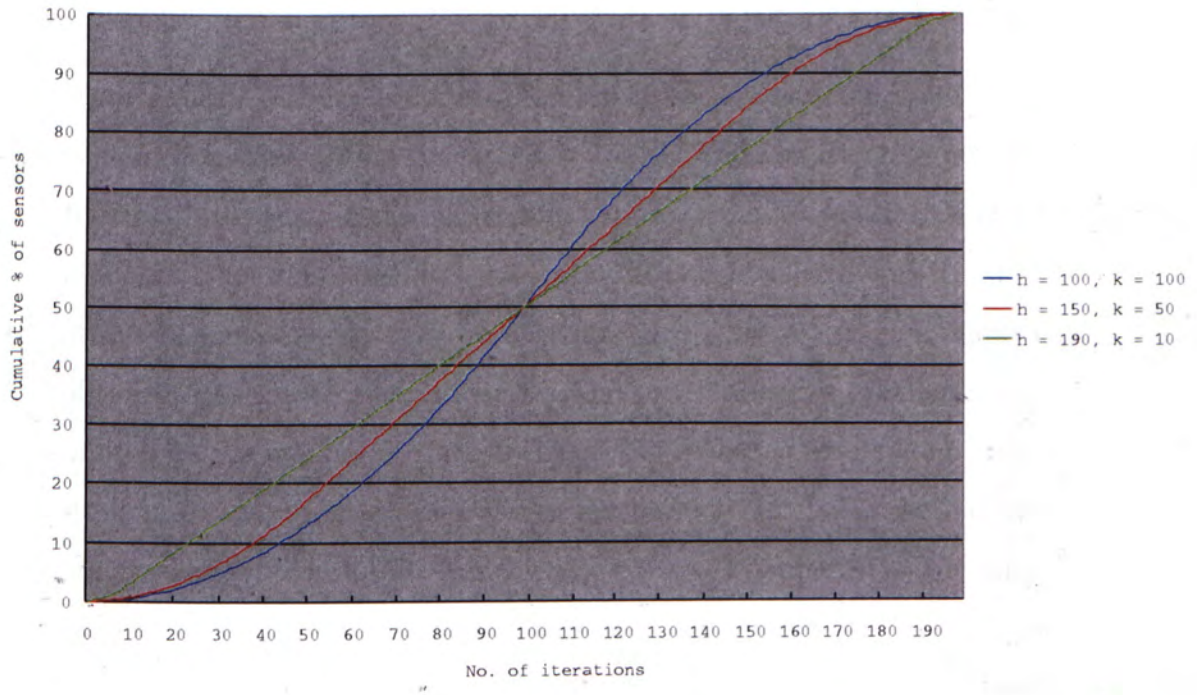
In the first set of simulations, we measured the construction speed of distributed PS data cubes in sensor networks by comparing the percentage of sensor nodes with their prefix sums and prefix averages computed against time. In each simulation, the relation between the width h and the length k of the gird-like sensor networks was $h + k = 200$. The simulation results are shown in Fig.3.11.

Now we can compare the construction speeds of distributed PS data cubes in several different grid forms. From Fig.3.11(a) to Fig.3.11(b), three main properties were found.

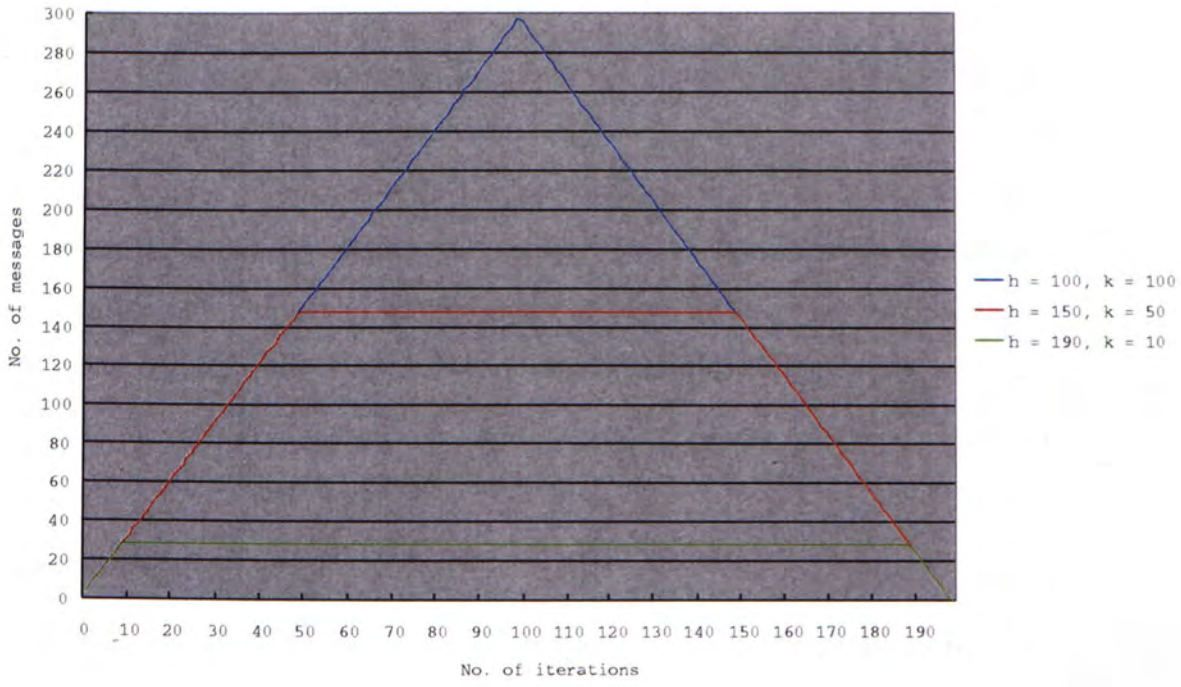
1. *The number of sensors with their prefix sums and prefix averages computed increased linearly in each iteration and attained a maximum.*
   The number of sensor nodes with the prefix sums and prefix averages computed increased linearly in each iteration, then it attained a maximum. It is because the number of sensors covered by the construction algorithm increased by one in each time unit (refer to Fig.3.6) during this period of time. When this value reached $\min(h, k)$, it is limited by the smaller one among the number of rows and the number of columns in the grid of sensor. Therefore, the value cannot increase anymore.


No. of sensors
No. of iterations
h = 100, k = 100
h = 150, k = 50
h = 190, k = 10

(a)


Cumulative % of sensors
No. of iterations
h = 100, k = 100
h = 150, k = 50
h = 190, k = 10

(b)



(c)

Figure 3.11: Construction time and network traffic against different h and k

2. *After attaining a maximum, the number of sensors with the prefix sums computed levelled off for the cases* $h = 150$, $k = 50$ *and* $h = 190$, $k = 10$. In Fig.3.11(a) the increase in sensors with their prefix values computed levelled off for the cases $h = 150$, $k = 50$ and $h = 190$, $k = 10$ after attaining a maximum. It is because after $k - 1$ iterations, the number of sensor nodes with the prefix sums and prefix averages calculated was limited by the number of rows in the grid of sensor. This remained constant until the number of iterations reached $h - 1$. This case did not appear when $h = k = 100$. It is because $t = h - 1$ happened at the same time when $t = k - 1$ occurred. Therefore, the number of sensor nodes with the prefix sums calculated was never limited by the number of rows in the grid.

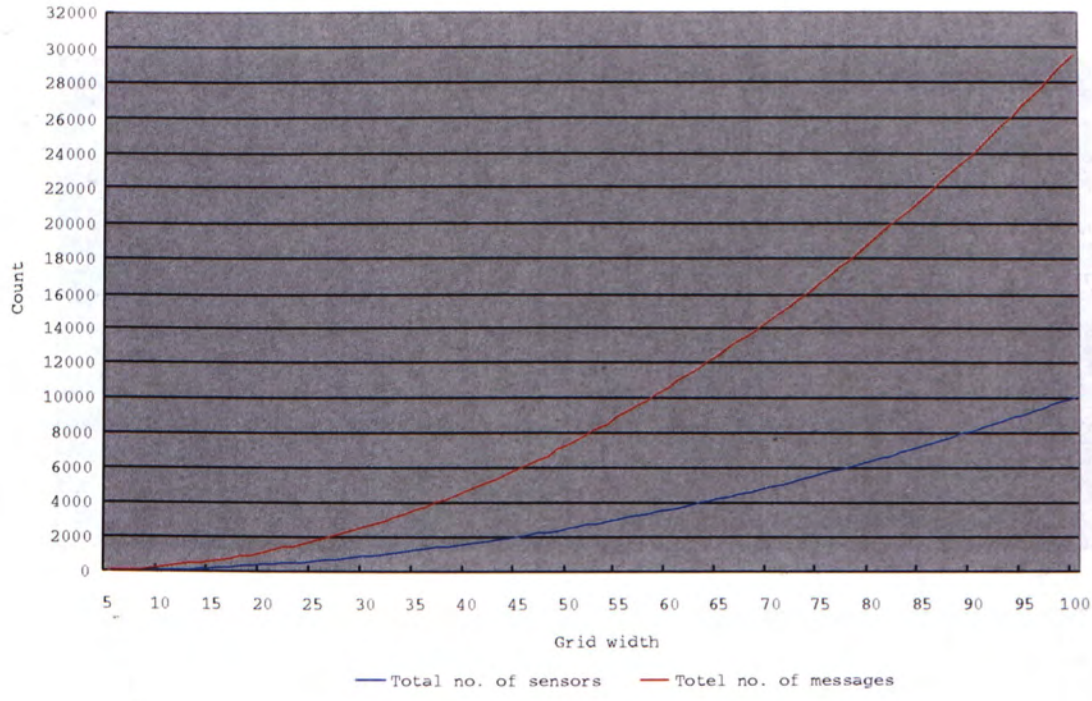
3. *After attaining a maximum, the number of sensors completed decreased linearly until the end of execution.*
   The number of sensors with the prefix sums computed began to drop when the number of iterations exceeded $h - 1$. The decrease was linear because the number of sensor nodes covered by the construction algorithm decreased by 1 in each iteration, as shown in Fig.3.6.

Network Traffic

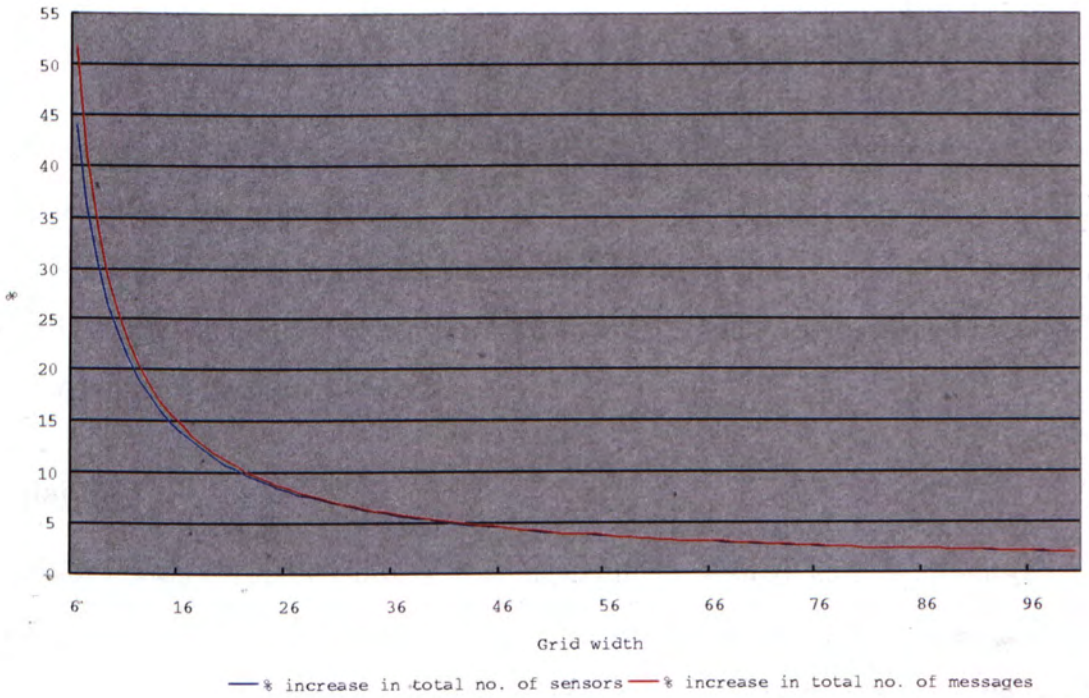
In order to study the network traffic during the construction of distributed PS data cubes, we carried out simulations to measure the total number of messages injected to each of the sensor networks by the sensors in every time unit. In the simulations, the width and the height of the grid-like sensor networks was once again $h + k = 200$. From Fig.3.11(c), three properties were observed.

1. *The number of messages injected into the network increased linearly and attained a maximum.*
   The network traffic increased linearly at the beginning of the distributed PS data cube construction process, as the number of sensors with the prefix sums and prefix averages computed increased linearly.

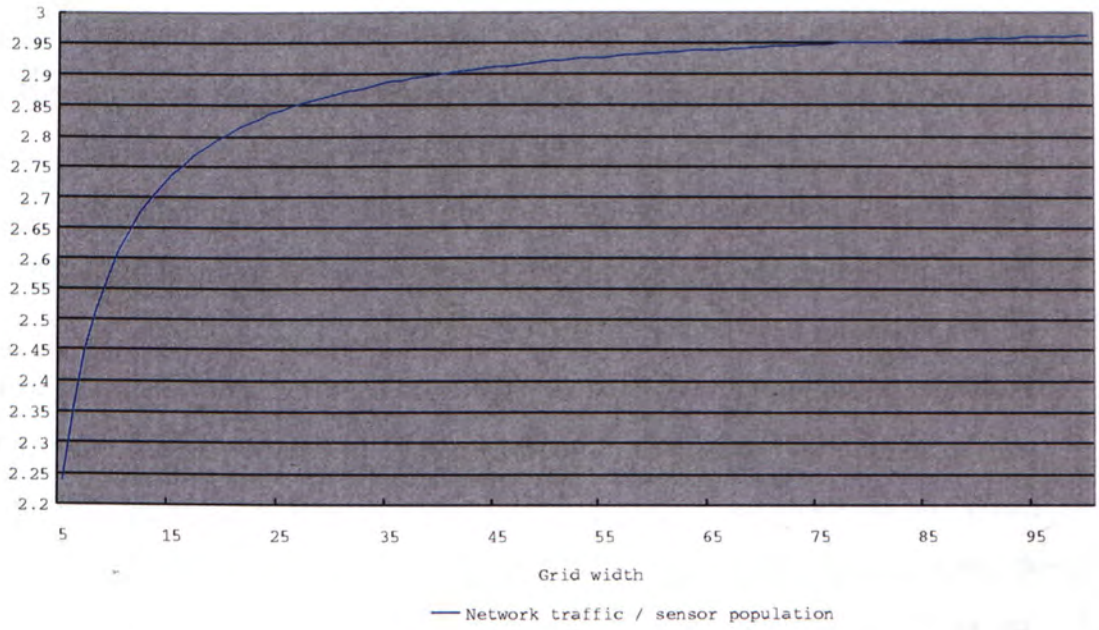
2. *When $h \neq k$, the network traffic levelled off for a period of time.*
   After $k - 1$ iterations, the number of sensors with the prefix sums and prefix averages calculated was limited by the number of rows in the cube. Hence the network traffic levelled off. This remained constant until the number of iterations reached $h - 1$. Since $t = k - 1$ and $t = h - 1$ happened at the same time unit for the case $h = k$, therefore the result


32000
30000
28000
26000
24000
22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Count
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Grid width
Total no. of sensors
Totel no. of messages

(a)


55
50
45
40
35
30
25
20
15
10
5
0
%
6 16 26 36 46 56 66 76 86 96
Grid width
% increase in total no. of sensors
% increase in total no. of messages

(b)



(c)

Figure 3.12: Network traffic against different network sizes

for the case $h = k = 100$ did not level off.

3. *After attaining a maximum, there was a linear drop in the network traffic until the end of execution.*

   This linear drop in the network traffic occurred when $t > h - 1$. It is because the number of sensors with the prefix sums and prefix averages computed decreased linearly.

Scalability

A sensor network usually consists of a huge number of small and cheap sensor nodes. Therefore, a good data retrieval algorithm for sensor networks should be scalable. In order to test the scalability of the proposed technique, especially the network traffic in terms of the total number of messages sent by the sensors,

simulations on 96 square grids of sensor network with different widths were performed. In the simulations, the sensor population sizes of the grids varied from 25 sensors to 10000 sensors and the total numbers of messages flowing in the networks during the construction of distributed PS data cubes were measured. The results are shown in Fig.3.12.

The simulation results show that the proposed distributed PS data cube construction technique scales well as the size of the square grid of sensor network increases. In Fig.3.12(a), we can see that the number of messages flowing through the network increased acceptably as the network population increased. When the sensor population increased from 25 to 10000, the number of messages flowing in the sensor network was only increased by about 30000. When Fig.3.12(b) is studied, we can observe that the percentage increase in network traffic decreased as the sensor network became larger, and the percentage increase of both the network traffic and the sensor population were approximately the same and they converged to a limit. In Fig.3.12(c), it can also be observed that the network traffic - sensor population ratio converged to 3 as the sensor population increased. That means each sensor sent 3 messages on average. From this observation, we can know that as the sensor network scales large, the network traffic in the sensor network is bounded by three times of the sensor population. With such a sharp bound, we can see that the proposed distributed PS data cube technique is scalable.

3.3 Distributed Local Prefix Sum (LPS) Data Cube

Though queries can be answered quickly using a distributed PS data cube, updating a distributed PS data cube is quite expensive. In the worst case, updating a distributed PS data cube can take as much time as rebuilding a new distributed PS data cube. This observation matches the findings regarding to the update cost of a traditional PS data cube [26].

The drawback of distributed PS data cube is that it may take a very long time for an update on a sensor locating at the upper left corner of the cube to be propagated to the sensors at the bottom right corner, especially when the distributed PS data cube is huge. In the worst case it takes linear time for a distributed PS data cube to be updated. For large scale sensor networks containing hundred thousands of sensors, it may be too expensive for distributed PS data cubes to be updated.

A straight forward solution to overcome the problem is to keep every distributed PS data cube small. This sounds impossible for huge size sensor networks, but actually it is possible. The secret is to partition any grid of sensor network into a number of smaller *blocks*, such that a distributed PS data cube is built in each block. The resultant distributed data cube would then be a *distributed local prefix sum (LPS) data cube.* Notice that distributed LPS data cube is inspired by the use of local prefix sum data cube [26, 28] in the traditional database systems.

In Fig.3.13(a), a distributed PS data cube is shown. Since it may take quite a long time to update the distributed PS data cube, we can partition the distributed data cube into blocks. Fig.3.13(b) is an example such that the grid is divided into four blocks, and the grid becomes a distributed LPS data cube with block size equal to 3. Since the length and the width of each block are only half of those of the original grid, the construction time and update

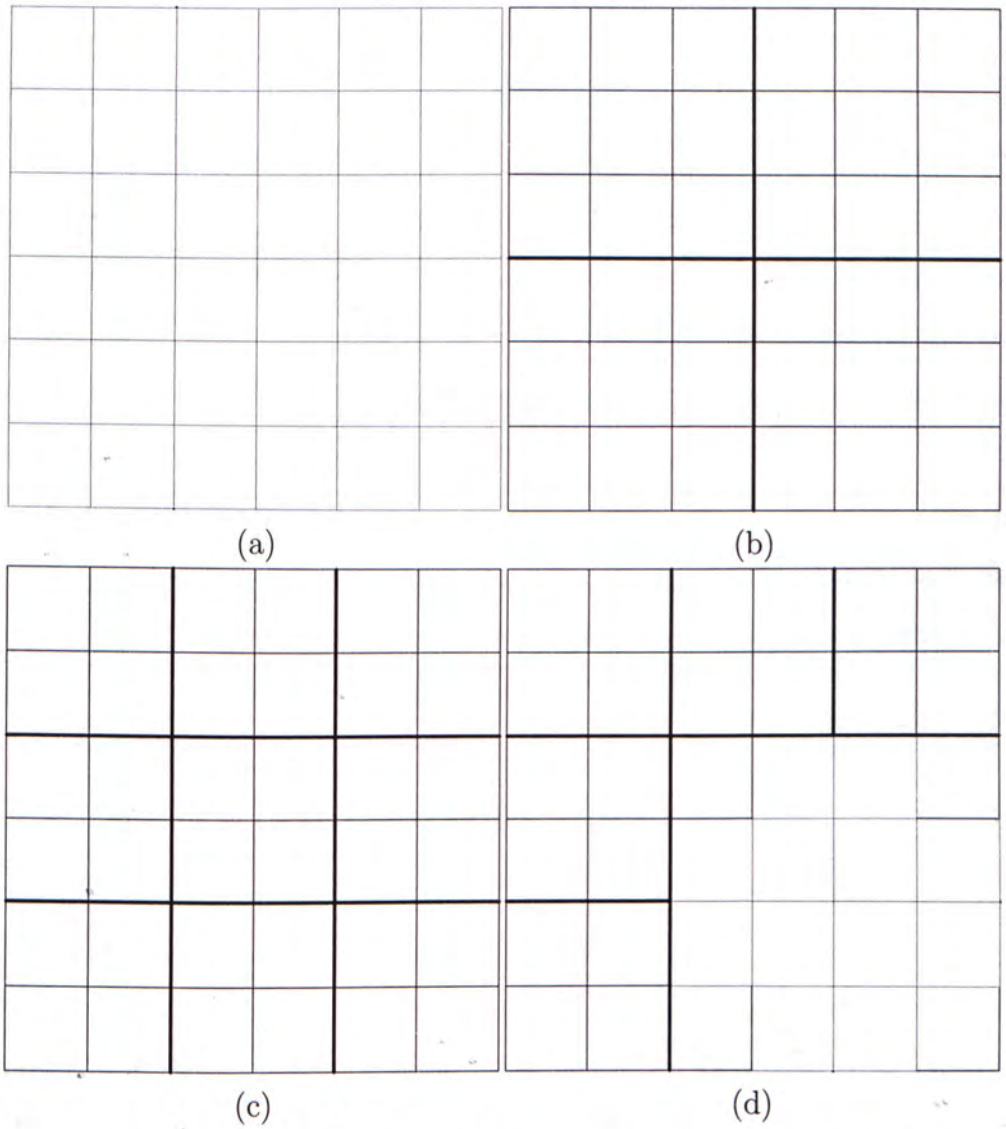


Figure 3.13: (a) A distributed PS data cube; (b) A distributed LPS data cube with block size = 3; (c) A distributed LPS data cube with block size = 2; (d) A distributed LPS data cube with different block sizes.

time of the whole distributed data cube can be halved. However, as a tradeoff, the querying time would be increased. It is because we need to query more sensors in order to obtain the data aggregate of a region, if the region covers a large number of blocks.

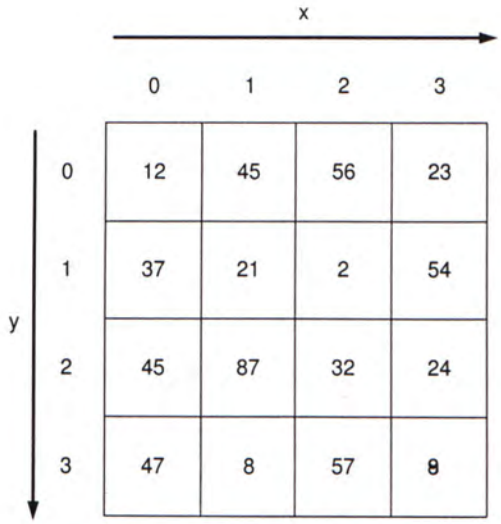
Fig.3.13(c) is another distributed LPS data cube with block size equal to 2. Its construction and update time are be just $\frac{2}{3}$ of those of Fig.3.13(b), but with a higher querying cost. According to [26], the blocks in an LPS data cube do not need to have the same block size. Fig.3.13(d) shows a distributed LPS data cube with different block sizes. However according to [26], the worst case update cost of an LPS data cube can be minimized if all blocks are of the same size. Therefore, in the rest of this thesis it is assumed that the blocks of any distributed LPS data cube are of the same block size.

An intuitive illustration of the proposed distributed LPS data cube has been given. However, distributed LPS data cube is originated from LPS data cube. So it is necessary to understand the principle of LPS data cube before beginning the discussion on distributed LPS data cube.

3.3.1 Local Prefix Sum Data Cube

Local prefix sum (LPS) data cube [26, 28] is a variation of prefix sum (PS) data cube. Since the update cost of PS data cube is relatively high, people then improved the data cube by partitioning it into a number of blocks and a PS data cube is built in each block. Each cell in an LPS data cube keeps the prefix sum of those cells of the source grid corresponding to its block (readers may refer to Definition 1 for the definition of prefix sum).

An example is shown in Fig.3.14. Fig.3.14(a) is the source data grid, and Fig.3.14(c) is an LPS data cube of the source data grid with block size equal to 2. Consider the block at the bottom left corner. Each cell in the block stores the prefix sum of those values in the grid shown in Fig.3.14(a), whose locations



(a)

	0	1	2	3
0	12	57	113	136
1	49	115	173	250
2	9	247	337	438
3	141	302	449	648

(b)

	0	1	2	3
0	12	57	56	79
1	49	115	58	135
2	45	132	32	56
3	9	187	89	211

(c)

Figure 3.14: (a) The source data grid; (b) The PS data cube of the source grid; (c) The LPS data cube of the source grid with block size = 2.

are covered by the block. For instance, the cell at $(1,3)$ of the LPS data cube stores the sum of values of those cells at $(0,2)$, $(1,2)$, $(0,3)$, and $(1,3)$. As a result, the prefix sum stored at the cell at $(1,3)$ is equal to 45+87+47+8 = 187.

Here is the formal definition of LPS data cube as introduced in [26,28]:

Definition 6 A 2-D local prefix sum (LPS) data cube is a 2-D grid which is partitioned into a number of square blocks b_m with block sizes s_m, $0 \leq m \leq M-1$. In each b_m, a PS data cube is built.

□

Describing the size of a block b_m by its block size s_m may cause ambiguity on the number of cells residing in b_m, since s_m may mean the number of cells in b_m or the width of b_m. From [26], we can define s_m without ambiguity:

Definition 7 The block size s_m of any block b_m of an LPS data cube refers to the width of the square block b_m. In any case, s_m is the number of cells that composes the width of b_m.

□

From Definition 7, we can know the number of cells $CellPop(b_m)$ in a block b_m given its block size s_m:

$$CellPop(b_m) = s_m^2 \tag{3.10}$$

From Definition 4, we know whether a cell c is in a region $e : f$. Similarly, we also need to define when a cell c is in a block b_m. Notice that e_{b_m} and f_{b_m} are the anchor and the endpoint of b_m respectively.

Definition 8 A cell c at (x_c, y_c) is in b_m if and only if $x_{e_{b_m}} \leq x_c \leq x_{f_{b_m}}$ and $y_{e_{b_m}} \leq y_c \leq y_{f_{b_m}}$.

□

Finally, a block b_m may or may not intersect with a region $e : f$, as defined as follows:

Definition 9 A block b_m is completely covered by a region $e : f$ if and only if every cell c_i in b_m is also in $e : f$.

□

Definition 10 A block b_m is not covered by a region $e : f$ if and only if every cell c_i in b_m is not in $e : f$.

□

Definition 11 A block b_m is partially covered by a region $e : f$ if and only if it is covered by $e : f$, but not completely covered by $e : f$.

□

3.3.2 Notations

In Section 3.2.2, we have defined a set of notations for the explanation of the principle of PS data cube. Since LPS data cube originates from PS data cube, hence the notations used before can still be adopted. Notations that are defined previously include the anchor e and endpoint f of any rectangular region $e : f$, the aggregate sum $Sum(e : f)$ of the region $e : f$, the sensor value $v(i)$ of sensor i, and its prefix sum $s(i)$. Readers are encouraged to read Definitions 1 to 5.

Finally, for an LPS data cube with M blocks b_m with block size s_m, $0 \leq m \leq M - 1$, we assume the anchor and endpoint of any block b_m to be e_{b_m} and f_{b_m} respectively.

3.3.3 Querying an LPS Data Cube

When we want to issue a query to find the data aggregate of a region in a grid of sensor network, we need to specify the region $e : f$ to be queried. We can imagine that the region $e : f$ will cover some blocks b_m completely or partially if all or some of the cells in b_m are also in $e : f$. Though regions and blocks are different terms describing collections of cells, we should try to relate them for the sake of further discussion. Here we formulate the rules for determining whether a block b_m is completely or partially covered by a region $e : f$ as follows.

Lemma 6 In an LPS data cube, a block b_m is completely covered by a region $e : f$ if and only if all of the following conditions are satisfied:

$$\begin{cases} x_e \leq x_{e_{b_m}} \\ y_e \leq y_{e_{b_m}} \\ x_f \geq x_{f_{b_m}} \\ y_e \geq y_{e_{b_m}} \end{cases}$$

Proof of Lemma 6 The four conditions ensure that every cell c_i in block b_m resides in $e : f$. The conditions can be visualized by Fig.3.15.

Assume that all the four conditions are true but b_m is not completely covered by $e : f$. Consider a cell c_i in b_m. Since c_i is in b_m, by Definition 8, $x_{e_{b_m}} \leq x_{c_i} \leq x_{f_{b_m}}$ and $y_{e_{b_m}} \leq y_{c_i} \leq y_{f_{b_m}}$. At the same time, by the four conditions the following inequalities can be obtained:

$$x_e \leq x_{e_{b_m}} \leq x_{f_{b_m}} \leq x_f \tag{3.11}$$

$$y_e \leq y_{e_{b_m}} \leq y_{f_{b_m}} \leq y_e \tag{3.12}$$

As a result, we know that $x_e \leq x_{c_i} \leq x_f$ and $y_e \leq y_{c_i} \leq y_f$. Therefore by Definition 4, every cell c_i in b_m is also in $e : f$. By Definition 9, b_m is completely

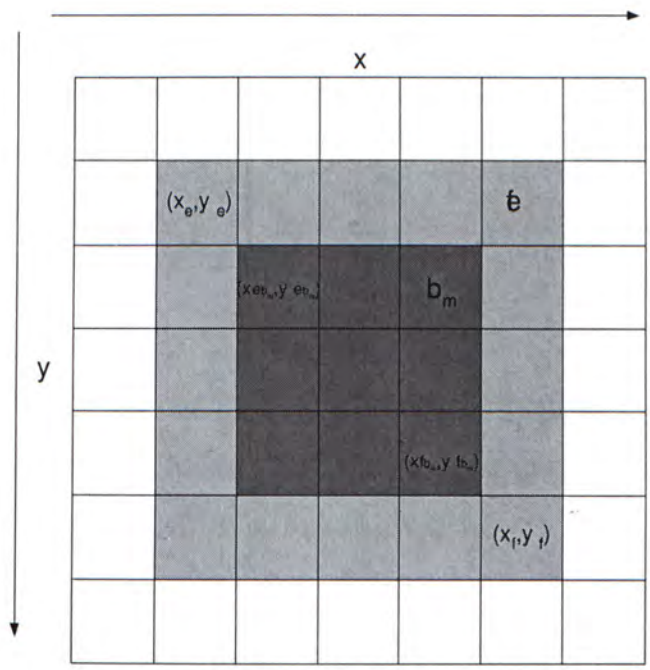


Figure 3.15: A block in a region.

covered by $e : f$. From the contrary result, b_m is completely covered by a region $e : f$ if all of the four conditions are satisfied.

Now assume that b_m is completely covered by $e : f$, but not all of the four conditions are true. It implies that there exists at least one cell c_i which is in b_m but not in $e : f$. Hence c_i is not covered by $e : f$. As c_i is a part of b_m, and c_i is not covered by $e : f$, by Definition 10, b_m is not completely covered by $e : f$. Due to the contrary result, b_m is completely covered by a region $e : f$ only if all of the four conditions are satisfied.

□

A block b_m can be partially covered by a region $e : f$ in too many cases. Hence instead of finding out all conditions for b_m to be partially covered by $e : f$, we would better find out the conditions where b_m is not covered by $e : f$. Then the conditions for b_m to be partially covered can be known.

Lemma 7 In an LPS data cube, a block b_m is not covered by a region $e : f$ if and only if any of the following conditions is satisfied:

$$\begin{cases} x_e > x_{f_{b_m}} \\ x_f < x_{e_{b_m}} \\ y_e > y_{f_{b_m}} \\ y_f < y_{e_{b_m}} \end{cases}$$

Proof of Lemma 7 Assume that b_m is covered by a region $e : f$ and at least one of the four conditions holds. Here b_m being covered by $e : f$ means that there exists at least one cell c_i in b_m which is also in $e : f$. By Definition 4, $x_e \leq x_{c_i} \leq x_f$ and $y_e \leq y_{c_i} \leq y_f$. At the same time, by Definition 8, $x_{e_{b_m}} \leq x_{c_i} \leq x_{f_{b_m}}$ and $y_{e_{b_m}} \leq y_{c_i} \leq y_{f_{b_m}}$. From these inequalities we can draw the following conclusion:

$$\begin{cases} x_e \leq x_{c_i} \leq x_{f_{b_m}} \Rightarrow x_e \leq x_{f_{b_m}} \\ x_f \geq x_{c_i} \geq x_{e_{b_m}} \Rightarrow x_f \geq x_{e_{b_m}} \\ y_e \leq y_{c_i} \leq y_{f_{b_m}} \Rightarrow y_e \leq y_{f_{b_m}} \\ y_f \geq y_{c_i} \geq y_{e_{b_m}} \Rightarrow y_f \geq y_{e_{b_m}} \end{cases}$$

It is contrary to the assumption that at least one of the four conditions holds.

Now assume that b_m is not covered by $e : f$ and none of the four conditions holds. Then it is likely to exist a cell c_i in b_m which satisfies the following conditions:

$$\begin{cases} x_e \leq x_{f_{b_m}} \Rightarrow x_e \leq x_{c_i} \leq x_{f_{b_m}} \\ x_f \geq x_{e_{b_m}} \Rightarrow x_f \geq x_{c_i} \geq x_{e_{b_m}} \\ y_e \leq y_{f_{b_m}} \Rightarrow y_e \leq y_{c_i} \leq y_{f_{b_m}} \\ y_f \geq y_{e_{b_m}} \Rightarrow y_f \geq y_{c_i} \geq y_{e_{b_m}} \end{cases}$$

By Definition 8, $x_{e_{b_m}} \leq x_{c_i} \leq x_{f_{b_m}}$ and $y_{e_{b_m}} \leq y_{c_i} \leq y_{f_{b_m}}$. At the same time, by Definition 4, $x_e \leq x_{c_i} \leq x_f$ and $y_e \leq y_{c_i} \leq y_f$. So c_i is also in $e : f$, violating the assumption that b_m is not covered by $e : f$.

Therefore, b_m is not covered by a region $e : f$ if and only if any of the four conditions is satisfied.

□

From Lemma 6 and Lemma 7, we know the condition for b_m to be partially covered by $e : f$:

Lemma 8 In an LPS data cube, a block b_m is partially covered by a region $e : f$ if and only if at least one of the following conditions is not satisfied:

$$\begin{cases} x_e \leq x_{e_{b_m}} \\ y_e \leq y_{e_{b_m}} \\ x_f \geq x_{f_{b_m}} \\ y_e \geq y_{e_{b_m}} \end{cases}$$

and none of the following conditions is satisfied:

$$\begin{cases} x_e > x_{f_{b_m}} \\ x_f < x_{e_{b_m}} \\ y_e > y_{f_{b_m}} \\ y_f < y_{e_{b_m}} \end{cases}$$

Proof of Lemma 8 Simply a result of Definition 11, Lemma 6, and Lemma 7.

□

Querying a PS data cube, as described before, is simple. By Lemma 1, we know that the prefix sums of at most four cells suffice for the calculation of the aggregate sum of any rectangular region. However, querying an LPS data cube is, in contrary, more complicated. It is because an LPS data cube is partitioned into blocks. For those blocks covered by the region where the data aggregate is wanted, some of them may be completely covered by the region while other may only be partially covered. We need to distinguish these two types of blocks before queries are issued. It is because a block is equivalent to a PS data cube, so for those blocks completely covered by the region in interest, we only need to query the prefix values held by the endpoints of the blocks. Otherwise, we need to query at most four sensors in order to get the aggregate of a partially covered block.

Since we know the block size of the LPS data cube and the region to be queried in advance, we can distinguish the two types of blocks by applying Lemma 6 and Lemma 8. Once the two types of blocks are distinguished, we can obtain the desired data aggregate by treating each block as a PS data cube, and applying the PS data cube querying algorithm on it. More specifically, Lemma 1 can be applied. To show the idea mathematically, assume that b_m are the blocks covered by the region $e : f$, $0 \leq m \leq M-1$, the aggregate sum $Sum(e : f)$ of $e : f$ equals:

$$Sum(e : f) = \sum Sum(e_{b'_m} : f_{b'_m}) \tag{3.13}$$

where $x_{e_{b'_m}} = \max(x_e, x_{e_{b_m}})$, $y_{e_{b'_m}} = \max(y_e, y_{e_{b_m}})$, $x_{f_{b'_m}} = \min(x_f, x_{f_{b_m}})$, $y_{f_{b'_m}} = \min(y_f, y_{f_{b_m}})$. The region $e_{b'_m} : f_{b'_m}$ represents the intersection between b_m and $e : f$, and $Sum(e_{b'_m} : f_{b'_m})$ is the aggregate sum of that intersecting region.

Since we need to know the data aggregate of every block which can be either completely or partially covered by a region, the number of query messages issued to an LPS data cube needed would be significantly larger than that

for a PS data cube. Therefore, whether a PS data cube or an LPS data cube should be used depends on the conditions and the requirements of the application where a data cube is to be built.

3.3.4 Building Distributed LPS Data Cube

We have addressed the working principle of LPS data cubes, the dependency between PS data cube and LPS data cube, and how aggregate sums can be queried from an LPS data cube. Now we are going to show how a distributed LPS data cube can be built in a grid of sensor network.

The assumptions we make here are similar to those we made when we discussed distributed PS data cubes. We suppose that the sensor network has a grid-like topology such that each cell contains a sensor. This can be achieved by adopting the GAF algorithm [29]. Besides that, each sensor i stores its sensor value $v(i)$ and prefix sum $s(i)$ in its memory. It is also assumed that each sensor can communicate with its neighbors and hence the sensor can broadcast messages to them.

Notice that, unlike the case for a distributed PS data cube, prefix averages are not stored in a distributed LPS data cube. Let us explain the reason by considering how the aggregate average of a region can be obtained using an LPS data cube. In fact, there are two ways to do so. First of all, we can obtain the prefix averages of the blocks covered by the region, then restore the aggregate product of each block. After that we need to calculate the total product of the aggregate products. Finally we need to divide the total product by the number of cells in the region. On the other hand, we can obtain the aggregate sum of the region. Then we can get the aggregate average by simply dividing the aggregate sum by the number of cells in the region. Obviously, the first way requires much more calculations than the second one. Hence we do not even keep the prefix averages of sensors in a distributed LPS data cube.

Algorithm 3 Maintain Prefix Sum in a Local Prefix Sum Data Cube

terminate = FALSE
$l(i)$ = empty, $u(i)$ = empty, $d(i)$ = empty

repeat
  if $s(j)$ from sensor j is received **then**
    if j is on the left of i **then**
      $l(i) = s(j)$
    end if
    if j is on top of i **then**
      $u(i) = s(j)$
    end if
    if j is on the upper left of i **then**
      $d(i) = s(j)$
    end if
    if $l(i)$, $u(i)$ and $d(i)$ are not empty **then**
      $s(i) = v(i) + l(i) + u(i) - d(i)$
      terminate = TRUE
    end if
  end if
until terminate = TRUE

if $x_i \neq x_{f_{b_m}}$ **then**
  // i is not at the right boundary of the block it locates
  i sends $s(i)$ to j_1 where j_1 is on the right of i
end if

if $y_i \neq y_{f_{b_m}}$ **then**
  // i is not at the bottom boundary of the block it locates
  i sends $s(i)$ to j_2 where j_2 is at the bottom of i
end if

if $i \neq f_{b_m}$ **then**
  // i is not the endpoint of the block it locates
  i sends $s(i)$ to j_3 where j_3 is at the bottom right of i
end if

Maintaining Prefix Sum

The construction algorithm of distributed LPS data cubes, and hence the maintenance of the prefix sums of sensors, is also based on Lemma 2. Consider a grid-like 2-D sensor network, in which each node i maintains a prefix sum $s(i)$, its own measurement $v(i)$, and three variables, namely $u(i)$, $l(i)$ and $d(i)$, where $u(i)$, $l(i)$ and $d(i)$ store the prefix sums received from the upper, left, and the upper left neighbors (i.e. sensors b, a and c respectively for sensor d in Fig.3.4). At time $t = 0$, the anchor e_{b_m} of every block b_m sets $s(e_{b_m}) = v(e_{b_m})$. Then it broadcasts $s(e_{b_m})$ to its neighbors. Once a sensor i receives the prefix sums from its upper, left and upper left neighbors, it will update the variables $u(i)$, $l(i)$, and $d(i)$ respectively according to Algorithm 3. After that it can compute its own prefix sum, and then broadcast the prefix sum to its neighbors. Eventually as Algorithm 3 is followed, an LPS data cube can be constructed.

The main difference between Algorithm 1 and Algorithm 3 is that the awareness of blocks is introduced in Algorithm 3. Now the distributed LPS data cube construction algorithm starts at the anchors of all blocks instead of just the anchor of the whole data cube. Furthermore, the sensors at the right and the bottom boundaries of each block are prohibited to send their prefix sums to the sensors belonging to other blocks. Finally, the construction algorithm stops at the endpoints of all blocks.

3.3.5 Time Bounds

In [26], the construction cost, query cost, and update cost of a 1-D LPS data cube are summarized. However, the costs of a 2-D distributed LPS data cube may be different. Now we are going to study the construction cost and the update cost of a distributed LPS data cube.

Construction Time

The construction of a distributed LPS data cube is equivalent to the construction of a number of small PS data cubes. As a result, the construction time of the proposed distributed LPS data cube depends on the selected block size. However, before analyzing the time bound of the algorithm, it is worth studying the number of time units for individual sensors to know their prefix sums. Readers are reminded that we assume all blocks are of the same size s_c.

Lemma 9 In a distributed LPS data cube with the block size s_c, assume that the time bound for a sensor to calculate and broadcast its prefix sum, and for the broadcasted prefix sum to be received by all its neighbors is 1. At time t, the prefix sums of all the nodes locating at (x, y), such that $x \pmod{s_c} + y \pmod{s_c} = t$, where $\pmod{s_c}$ is the modulus operator with divisor s_c, will be ready.

Proof of Lemma 9 In a distributed LPS data cube, blocks are formed from $(0, 0)$ horizontally and vertically with block size s_c. Therefore, those cells c at (x, y) with $x \pmod{s_c} = 0$ and $y \pmod{s_c} = 0$ will be the anchors of the blocks they reside. On the other hand, cells c at (x, y) with $x \pmod{s_c} = s_c - 1$ and $y \pmod{s_c} = s_c - 1$ are the endpoints. So now the scenario becomes constructing a number of distributed PS data cube in all blocks, with the construction process starting at the anchors of all blocks. By Lemma 3, at time t those cells c at (x, y) relative to the anchors of the blocks their belong where $x + y = t$ will know their prefix sum.

Consider a particular block b_m with anchor e_{b_m} at $(x_{e_{b_m}}, y_{e_{b_m}})$. At time t, the cell c at (x, y) relative to e_{b_m} knows its prefix sum. Assume that the real coordinates of c relative to $(0, 0)$ are (x_{real}, y_{real}). The relation between (x_{real}, y_{real}) and (x, y) is:

$$x_{real} = x_{e_{b_m}} + x \Rightarrow \quad x_{real} \pmod{s_c} = x \tag{3.14}$$

$$y_{real} = y_{e_{b_m}} + y \Rightarrow \quad y_{real} \pmod{s_c} = y \tag{3.15}$$

From Equation 3.14, Equation 3.15, and $x + y = t$, we know that at time t the prefix sums of all the sensor nodes with x, y-coordinates (x, y), such that $x \pmod{s_c} + y \pmod{s_c} = t$, will be ready.

□

Now we are going to show the time bound of the proposed distributed LPS data cube construction algorithm.

Lemma 10 In a sensor network with length $= h$ cells and width $= k$ cells, a distributed LPS data cube with the block size s_c can be completely constructed in $2s_c - 2$ units of time.

Proof of Lemma 10 Consider a particular block b_m with the endpoint f_{b_m} at $(x_{f_{b_m}}, y_{f_{b_m}})$. From Lemma 9, the prefix sum of f_{b_m} will be computed at time unit $t = x_{f_{b_m}} \pmod{s_c} + y_{f_{b_m}} \pmod{s_c}$. Since $x_{f_{b_m}} \pmod{s_c} + y_{f_{b_m}} \pmod{s_c} = s_c - 1 + s_c - 1 = 2s_c - 2$, f_{b_m} will know its prefix sum at $t = 2s_c - 2$, and so do the endpoints of other blocks. As a result, a distributed LPS data cube with block size s_c can be completely constructed in $2s_c - 2$ units of time.

□

From Lemma 10, we know that the time complexity of the proposed distributed LPS data cube construction algorithm $O(s_c)$, where s_c is the block size of the distributed LPS data cube.

Update Time

Since sensors may update their sensor values continuously until they run out of batteries, we need to handle sensor value updates properly. Similar to the case of a distributed PS data cube, the sensor value of a sensor may be included in the calculation of the prefix sums of some other sensors. Hence after any sensor has updated its sensor value, the update should be forwarded to other sensors locating at the bottom right corner of the same block.

If a sensor at the top left corner of a distributed PS data cube updates its sensor value, all other sensors will need to be updated. In contrast, in a distributed LPS data cube only the sensors in the same block of the sensor with its sensor value being changed need to be updated. It is because the cube is now divided into blocks, which are independent and do not affect one another. Actually, it is possible for blocks of sensors to be updated separately and asynchronously. It can be achieved by running Algorithm 3 in each block, by assuming the sensor being updated be the pseudo-anchor of the block. By doing so, we can know the time needed for a distributed LPS data cube to be updated.

Lemma 11 In a distributed LPS data cube with the block size s_c, any block b_m can be updated in $O(s_c)$ time.

Proof of Lemma 11 As we mentioned earlier, it is possible for blocks of sensors to be updated separately and asynchronously. It can be achieved by running Algorithm 3 in each block, by assuming the sensor be the pseudo-anchor of b_m. In the worst case, this pseudo-anchor is exactly the real anchor e_{b_m} of the block b_m. By Lemma 10, a distributed LPS data cube with block size s_c can be completely constructed in $2s_c - 2$ units of time. Hence the block of sensors can be updated in $2s_c - 2$ units of time. As a result, block b_m can be updated in $O(s_c)$ time.

□

3.3.6 Fast Aggregate Queries on Multiple Regions

Querying a distributed LPS data cube is similar to querying a distributed PS data cube, though it is more complicated. It is because a distributed LPS data cube is partitioned into blocks. Hence when issuing queries to sensors for their prefix sums, we need to know which blocks are partially or completely covered by the region we query on. Since we know the block size of the distributed LPS data cube, and the anchor and the endpoint of the region to be queried in advance, we can distinguish the two types of blocks by applying Lemma 6 and Lemma 8. Once the two types of blocks are distinguished, we can obtain the desired aggregate sum of the region by treating each block as a distributed PS data cube, and applying the distributed PS data cube querying algorithm on it. More specifically, Lemma 1 can be applied. To show the idea mathematically, assume that b_m are the blocks either completely or partially covered by the region $e : f$, $0 \leq m \leq M-1$, the aggregate sum $Sum(e : f)$ of $e : f$ equals:

$$Sum(e : f) = \sum Sum(e_{b'_m} : f_{b'_m}) \tag{3.16}$$

In Equation 3.16, we assume $x_{e_{b'_m}} = \max(x_e, x_{e_{b_m}})$, $y_{e_{b'_m}} = \max(y_e, y_{e_{b_m}})$, $x_{f_{b'_m}} = \min(x_f, x_{f_{b_m}})$, $y_{f_{b'_m}} = \min(y_f, y_{f_{b_m}})$ for each block b_m. The region $e_{b'_m} : f_{b'_m}$ represents the intersection between any block b_m and the region $e : f$, and $Sum(e_{b'_m} : f_{b'_m})$ is the aggregate sum of that overlapping area.

To compute for the aggregate average $Avg(e : f)$ of $e : f$, we need to obtain the aggregate sum $Sum(e : f)$ and the total number of sensor population $SenPop(e : f)$ of $e : f$. Then $Avg(e : f) = \frac{Sum(e:f)}{SenPop(e:f)}$.

Since we need to know the aggregate sum of every block either completely or partially covered by a region, the number of query messages issued to a distributed LPS data cube is significantly larger than that for the case of a

Distributed Data Cube	Construction Cost	Update Cost	Query Cost (rectangular regions)
Distributed PS Data Cube	$N-1$	$N-1$	4
Distributed LPS Data Cube	$2s_c-2$	$2s_c-2$	$4N_p+N_c$

Table 3.2: The worst case costs of the two distributed data cubes

distributed PS data cube. In the worst case, one query is needed for the retrieval of aggregate sum from each block completely covered by the region, and four queries are needed for the retrieval of aggregate sum from each block partially covered by the region. As a result, assume that N_c blocks are completely covered by the region and N_p blocks are only partially covered, the total queries required will be up to N_c plus four times of N_p. Therefore, the worst case query cost for a distributed LPS data cube can be much higher than that of a distributed PS data cube, especially when the block size is small. So whether a distributed PS data cube or a distributed LPS data cube should be used depends on the application in which a distributed data cube is needed. If we decide to use a distributed LPS data cube, we need to tune the block size carefully to obtain the optimal performance.

Table 3.2 summarizes the construction costs, update costs, and querying costs of distributed PS and LPS data cubes. Notice that in Table 3.2, N_c and N_p are the number of blocks completely and partially covered by the region being queried respectively, and s_c is the block size.

3.3.7 Simulation Results

We performed simulations to study the behaviours of distributed LPS data cubes. In this part, we will show the simulation results on the construction speed, network traffic, and scalability of distributed LPS data cubes.

Data Cube Construction Speed

We measured the construction speed of distributed LPS data cubes in sensor networks by comparing the percentage of sensor nodes with their prefix sums computed against time. In each simulation, the relation between the width h and the length k of the distributed LPS data cubes was $h + k = 200$. Furthermore, the block size is 10 so that each block contains 100 sensors. The simulation results are shown in Fig.3.16.

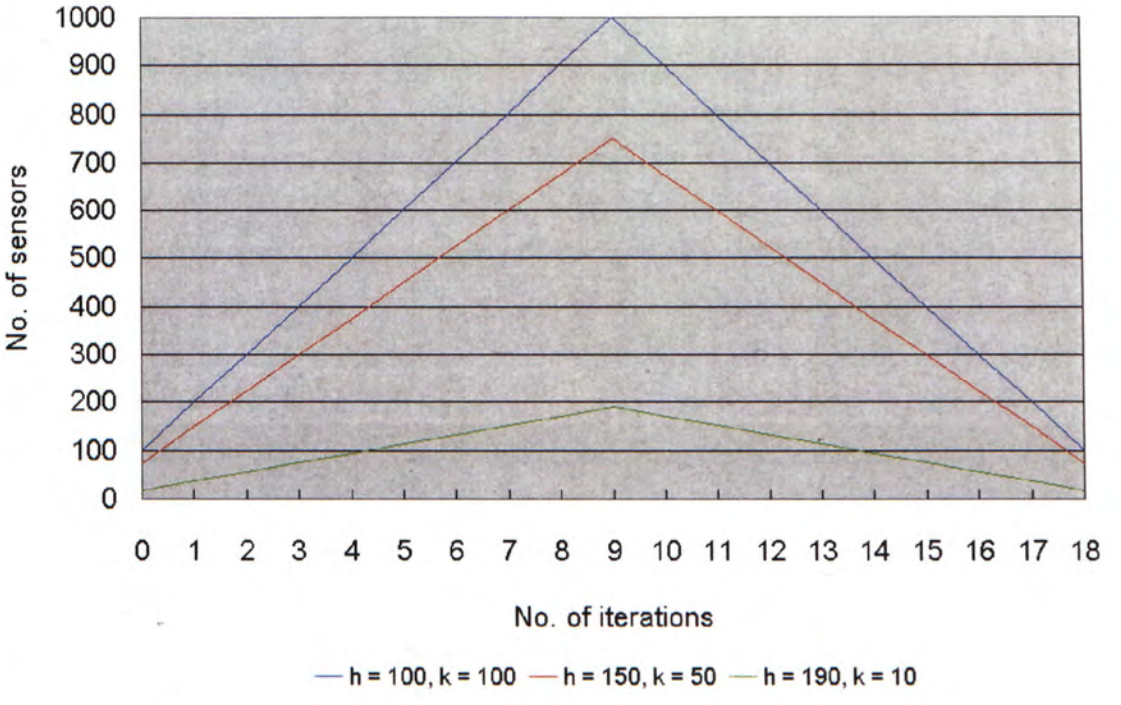
From the figures, four observations can be made:

1. *The number of sensors with their prefix sums computed in each iteration increased linearly and attained a maximum.*

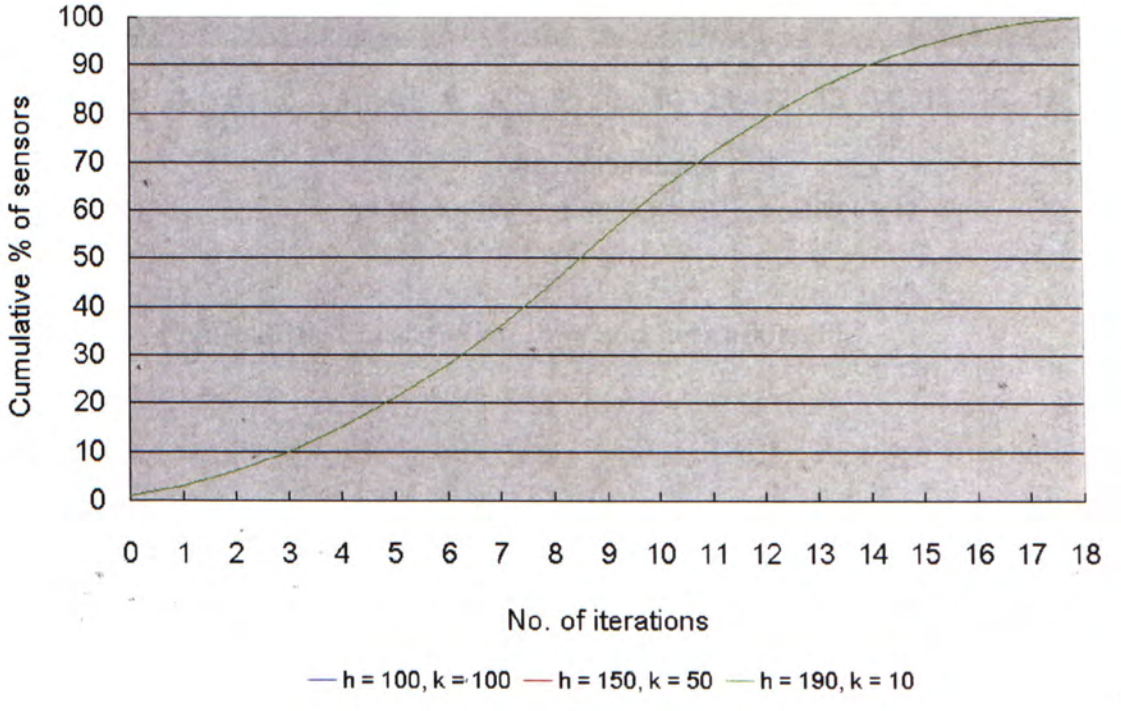
   In Fig.3.16(a), the number of sensor nodes with the prefix sums computed in each iteration increased linearly first, then it attained a maximum. It is because each block was a distributed PS data cube, and according to the result shown in Fig.3.11(a) the number of sensor nodes with the prefix sums computed in each iteration increased linearly. Hence for distributed LPS data cubes, the total number of sensors with the prefix sums computed increased linearly in each iteration and reached a maximum.

2. *After attaining a maximum, the number of sensors with the prefix sums computed did not level off*

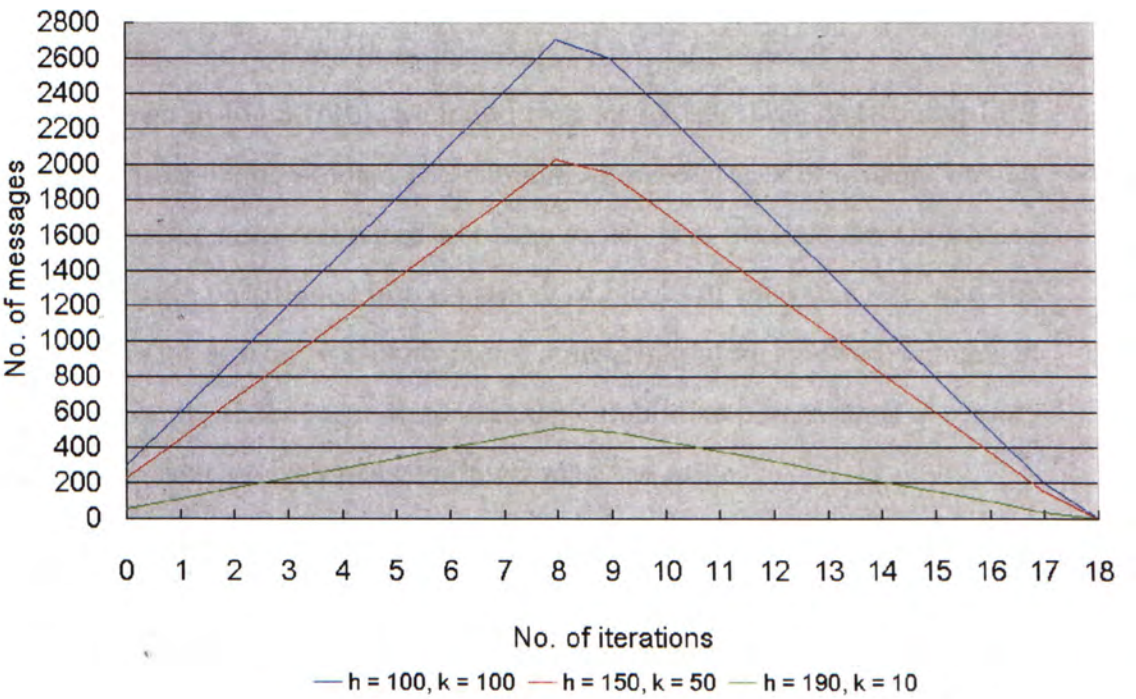
   Contrary to the results for distributed PS data cubes (Fig.3.11(a)), after attaining a maximum the number of sensors with the prefix sums computed did not level off for distributed LPS data cubes (Fig.3.11(a)). It is because each distributed LPS data cube was divided into a number of square blocks in which distributed PS data cubes were built. Since the blocks were square, according to the result for a square distributed PS data cube shown in Fig.3.16(a), the number of sensors with the prefix sums computed did not level off.


1000
900
800
700
600
500
400
300
200
100
0
No. of sensors
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
No. of iterations
h = 100, k = 100
h = 150, k = 50
h = 190, k = 10

(a)


100
90
80
70
60
50
40
30
20
10
0
Cumulative % of sensors
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
No. of iterations
h = 100, k = 100
h = 150, k = 50
h = 190, k = 10

(b)



(c)

Figure 3.16: Construction time and network traffic

3. *After attaining the maximum, the number of sensors completed decreased linearly until the end of execution.*

   The number of sensors with the prefix sums computed began to drop when the number of iterations exceeded $s_c - 1$, but not $h - 1$. It is because each block was a square distributed PS data cube and it followed the behaviours for a square distributed PS data cube as displayed in Fig.3.11(a).

4. *The cumulative percentages of sensors having their prefix sums computed were the same for the three distributed LPS data cubes.*

   As shown in Fig.3.16(b), we found that for all the three distributed LPS data cubes being studied the cumulative percentages of sensors having their prefix sums computed were the same. It is because the three data cubes were partitioned into a number of blocks of the same size, and the number of sensors with the prefix sums computed followed Lemma 9. As a result, at the same time unit the cumulative percentages of sensors having their prefix sums computed were the same.

Network Traffic

We studied the network traffic during the construction of distributed LPS data cubes by carrying out simulations to measure the total number of messages injected to the network by the sensors in each time unit. In the simulations, the width and the height of the distributed LPS data cubes was again related by $h + k = 200$ with block size 10. From Fig.3.16(c), three properties were found.

1. *The number of messages injected into the network increased linearly and attained a maximum.*

   The network traffic increased in the beginning, as the number of sensors with the prefix sums computed increased linearly.

2. *The network traffic did not level off after attaining a maximum.*
   Contrary to the results for distributed PS data cubes (Fig.3.11(c)), for distributed LPS data cubes, the number of messages in the network attained a maximum value, and then the value did not level off (Fig.3.16(c)). It is because the distributed LPS data cubes were divided into a number of square blocks in which distributed PS data cubes were built. Since the blocks were square, hence according to the result shown in Fig.3.11(c), the number of sensors with the prefix sums computed did not level off. Notice that before the network traffic dropped quickly, there was a slight drop in the network traffic. It is because the sensors at the boundaries of the blocks send less than three messages, and hence there was a slight drop in the total number of messages flowing in the sensor network.

3. *When $t \geq s_c - 1$ the value decreased linearly.*
   The network traffic decreased linearly because the number of sensors with their prefix sums computed decreased linearly.

Scalability

To study the scalability of distributed LPS data cube, we tested 96 distributed LPS data cubes with varying sizes. We found that a distributed LPS data cube with the same size of a distributed PS data cube scaled exactly the same, independent of the block size. Therefore, we can conclude that distributed LPS data cubes have the same performance, in terms of scalability, as that of a distributed PS data cube.

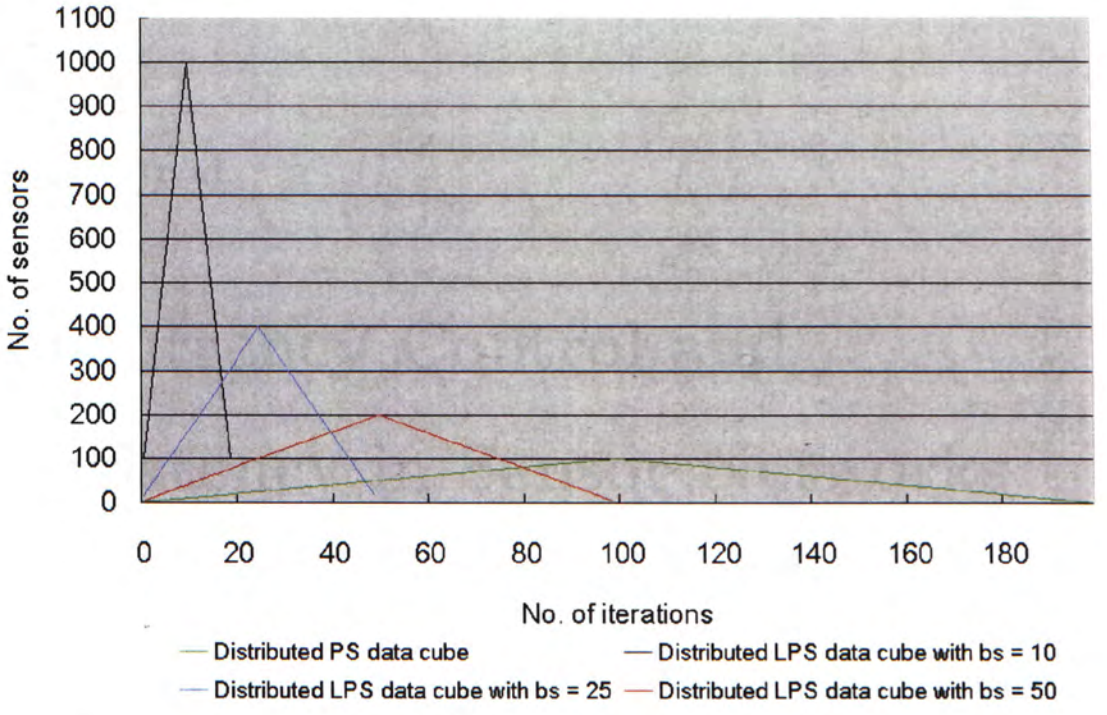
3.3.8 Distributed PS Data Cube Vs Distributed LPS Data Cube

Fig.3.17 shows the comparisons in the construction speeds and the network traffic incurred between a distributed PS data cube and three distributed LPS data cubes with varying block size. The distributed data cubes were all with $h = 100$, $k = 100$, and the block sizes of the three distributed LPS data cubes were 10, 25, and 50 respectively.

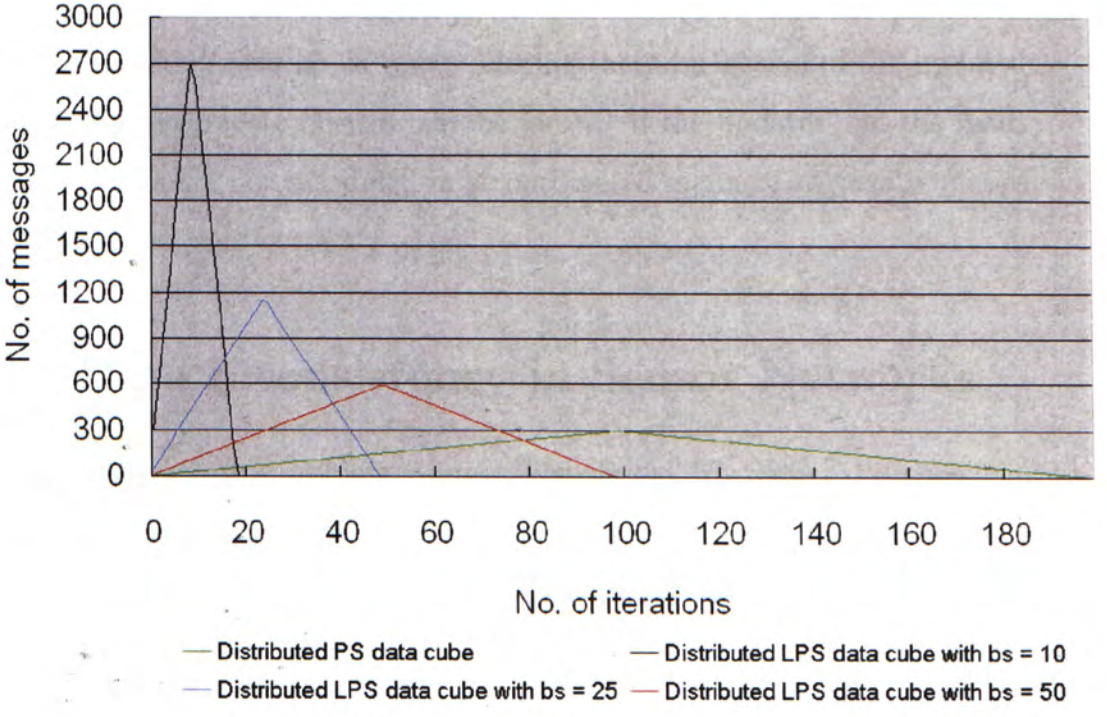
From Fig.3.17(a), we observe that all the three distributed LPS data cubes took less construction time than that of the distributed PS data cube did. For the three distributed LPS data cubes, the construction times were in the order $s_c = 10 < s_c = 25 < s_c = 50$.

On the other hand, the distributed LPS data cubes led to much busier network traffic than that of a distributed PS data cube (Fig.3.17(b)). For the three distributed LPS data cubes, the network traffic incurred was in the order $s_c = 10 > s_c = 25 > s_c = 50$.

As a conclusion, we should apply the suitable distributed data cube for an application. The choice depends heavily on the update frequency, query frequency, and network resources. If we decide to use a distributed LPS data cube, we may tune the block size until the behaviours match our requirements.



(a)



(b)

Figure 3.17: PS data cube Vs LPS data cube

Chapter 4

Concurrency Control and Consistency in Sensor Networks

The sensors in a sensor network may update their sensor values occasionally, and the updated sensor values should be diffused to all other sensors requiring them for the calculation of their prefix sums and prefix averages. These sensors will then re-calculate their prefix values accordingly, and propagate the new values to the other sensors. However, during the propagation of the updated sensor values, aggregate queries may be issued. If the updates and the reads of prefix values are not scheduled in a coordinated manner, aggregate queries may not be answered correctly.

4.1 Data Inconsistency in Sensor Networks

We can illustrate the problem using a distributed PS data cube shown in Fig.4.1. In the example we assume that the reading $v(i)$ of every sensor i is positive, and hence so does $s(i)$. Fig.4.1(a) shows the initial sensor values and the prefix sums. Assume that sensor b detects an environmental change and $v(b)$ is increased by 32. Since $s(b)$ includes $v(b)$, $s(b)$ is also updated in order to reflect the change in $v(b)$. In Fig.4.1(b) we can see that the updated values of $v(b)$ and $s(b)$, which are 38 and 50 respectively. Now as $s(b)$ has

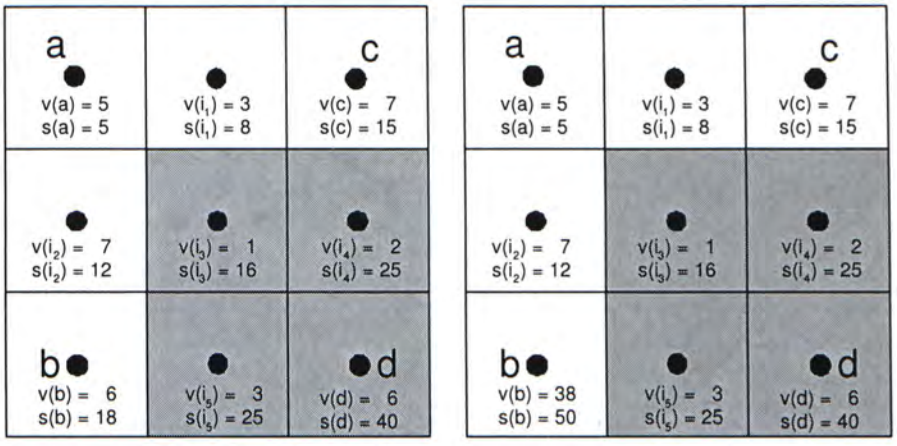


Figure 4.1: (a) A consistent distributed PS data cube; (b) An inconsistent distributed PS data cube

been updated, sensor b will propagate the new $s(b)$ to its neighbors for them to update their prefix sums accordingly. However, it takes time for the update to be propagated and there exists a period of time such that sensor b has updated its prefix sum but other sensors have not. At this moment, if someone issues a query on the aggregate sum of the shaded region, $s(a)$, $s(b)$, $s(c)$, and $s(d)$ will be returned and the aggregate sum is $s(d) - s(b) - s(c) + s(a) = 40 - 50 - 15 + 5 = -20$, which is negative. Recall that all sensor values are positive, so the aggregate sum of any region in the sensor network can never be negative. From this contradictory result, we see that the distributed PS data cube fails to answer the aggregate query correctly. It is because the prefix sums stored in the distributed PS data cube correspond to different time units and hence the distributed PS data cube becomes *inconsistent*.

In fact, we found that inconsistency does not only appear in distributed data cubes that we proposed, but it also happens in other existing hierarchical aggregation techniques. Let us illustrate the problem by Fig.4.2. In the figure, an aggregation tree like the one proposed in [10] is maintained in the sensor network for the in-network calculation of the aggregate sum of the sensors.

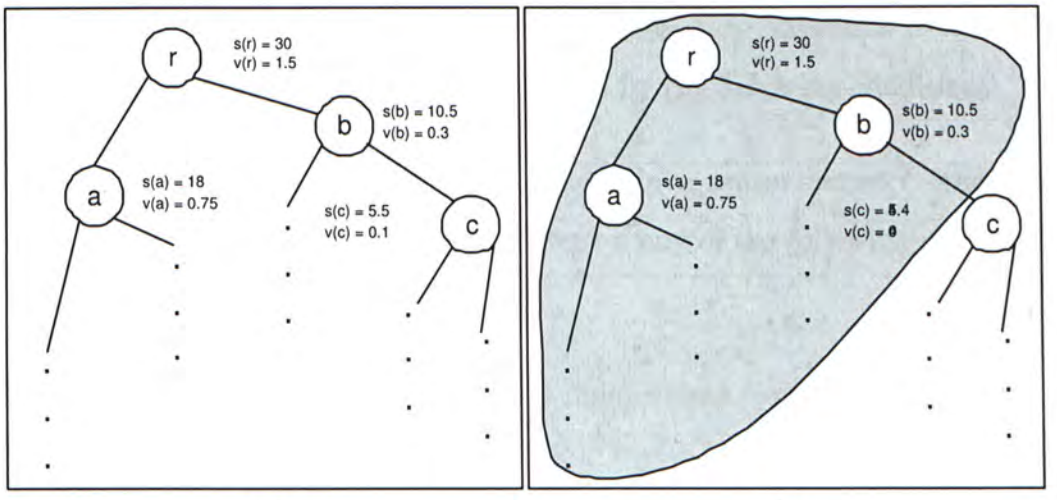


Figure 4.2: Data inconsistency due to data transmission latency and the concurrent execution of conflict operations

Each sensor i in the network holds two values, its sensor value $v(i)$ and the aggregate sum $s(i)$ of the sub-tree rooted at i. In the example, it is assumed that all the sensor values are positive. Fig.4.2(a) shows the initial sensor values and aggregate sums. At a particular moment, sensor c detects a sudden change in the environment and $v(c)$ increases from 0.1 to 40. Consequently, $s(c)$ increases to 45.4 (Fig.4.2(b)). Now since $v(c)$ is updated, the updated value should be forwarded to all predecessors of sensor c in order to reflect the change of $v(c)$. However, time is needed for the propagation of the updated value. Before the update reaches the root r, if we issue a query on $s(r)$ and another one on $s(c)$ for the aggregate sum of the sensor values of all sensors excluding those in the sub-tree rooted at sensor c (i.e. the shaded region), we will get 30 and 45.4 respectively. Finally the desired result can be obtained by $s(r) - s(c) = -15.4$, which is contradictory to the assumption that all sensor values are positive.

Based on the examples we showed above, we can determine if the data obtained from any two sensors are consistent by the following conditions:

Definition 12 For any two sensors a and b in a sensor network, the data retrieved from sensors a and b are consistent if any of the following conditions is satisfied:

- The data retrieved from sensor a is independent from the data retrieved from sensor b, and vice versa;
- The data retrieved from sensor b is dependent on the data retrieved from sensor a, and the data from sensor b includes the effect of the corresponding version of the data from sensor a.

□

Let us study the examples in Fig.4.1 and Fig.4.2 again, using Definition 12 as the rule to determine if the data in the examples are consistent.

In Fig.4.1, $s(d)$ is dependent on $s(b)$ since $s(d)$ must be updated after $s(b)$ has been updated. In Fig.4.1(a), $s(b)$ and $s(d)$ are consistent because $s(d)$ includes the effect of the corresponding $s(b)$. However $s(b)$ and $s(d)$ in Fig.4.1(b) are inconsistent because $s(d)$ does not include the the effect of $s(b)$, that is the increase in $s(b)$.

In Fig.4.2, $s(b)$ is dependent on $s(c)$ since $s(b)$ must be updated after $s(c)$ has been updated. $s(b)$ and $s(c)$ in Fig.4.2(a) are consistent because $s(b)$ includes the latest update of $s(c)$. However $s(b)$ and $s(c)$ in Fig.4.2(b) are inconsistent because $s(b)$ does not include the effect of the increase in $s(c)$.

4.2 Traditional Concurrency Control Protocols and Sensor Networks

We observe that the problem of data inconsistency exists in sensor network systems. As a consequence, we need some effective and efficient measures for the concurrency control of sensor networks. As we mentioned before, sensor networks are quite different from traditional distributed systems. So can traditional concurrency control protocols for distributed systems work in sensor networks?

In traditional distributed systems, two protocols are widely used for the concurrency control of read-only and read-write operations. They are two-phase locking protocol and time-stamp ordering protocol. They ensure the serializability of concurrent interleaving operations, and hence provide concurrency control to a distributed system. Readers are recommended to refer to Section 2.4 for more detail.

From Section 2.4.1, we know that the principle of two-phase locking protocol is to ensures transactions to be executed with a serialization order by granting locks so that the transactions can be serializable. However, deadlocks may occur when two or more transactions are waiting for the locks held by one another in a circular way. In this case, the system may hang because none of the transactions involved in a deadlock can be executed to completion. Deadlock detection and avoidance algorithms have been proposed, but they usually incur significant overheads. As a result, they are not suitable to be used in distributed systems like sensor network systems with limited resources.

Some people may think that timestamp ordering protocol fails to work in sensor networks. However, we found that it still works well in sensor network systems. It is because a timestamp is issued to a transaction once it is invoked, and hence the serialization order (i.e. the serializable execution order) of all transactions is pre-determined [46]. After that, every transaction is forced

to obey this order even with the presence of transmission latency and sensor mobility. The main point is that aged transactions cannot be executed after those recent transactions even if they are delayed, and hence data will not be modified in a inconsistent way.

To further improve the degree of concurrency, multiversion distributed systems are developed [46–53] such that more than one version of data are stored and more transactions can be served concurrently. In [52], a modular approach for the version control and concurrency control of distributed system is proposed. It allows read-only transaction to be executed independent of the implementation of the concurrency control protocol. We think that this modular approach can fulfill the requirements of sensor network systems when it is combined with a timestamp-based concurrency control protocol.

4.3 The Consistent Retrieval of Data from Distributed Data Cubes

A state-of-the-art solution for the concurrency control of distributed systems is proposed in [52]. It acts as a platform with the support of multiversion control so that both of the two-phase locking protocol and timestamp ordering protocol can be applied based on it. However, instead of quoting the work in [52], we will introduce a simple synchronous algorithm for the proposed distributed data cubes so that readers can understand the spirit of multiversion concurrency control protocol by studying our algorithm. The approach is synchronous not because there exists a common clock for the control of sensor operations. Instead, it is because only the sensor at the upper left corner of the whole distributed data cube can invoke the update process of the distributed data cube, and other sensors are updated only when they have received all the three prefix sums from the upper left neighbors.

Since multiversion concurrency control allows data to be read independently from the data update process, the synchronous protocol we introduce here will be a multiversion algorithm and more than one version of sensor values, prefix sums, and prefix averages are stored in each sensor. Notice that as the prefix average of any sensor is derived from its prefix sum, we only need to deal with the prefix sums of sensors and ensure that they are modified and read consistently. Furthermore, we will introduce the proposed algorithms by focusing on distributed PS data cubes. With slight modifications, the algorithms can also be applied on distributed LPS data cubes.

In Section 3.2.4, we have presented the algorithm for maintaining prefix sums in a grid of sensors (please refer to Algorithm 1). The synchronous algorithm we propose is still based on Algorithm 1, with the addition of the following policies to the distributed PS data cube construction and update process: At time $t = 0$, the sensor i at $(0, 0)$ of the whole distributed PS data cube can compute its prefix sum and prefix average, since it does not depend on other sensors for its prefix values. Then it needs to save $s(i)$ in its memory with a timestamp $ts(i)$, which is initially one, in order to answer queries in the future. After that, sensor i has to send $s(i)$ to its neighbors according to Algorithm 1. To invoke the update process of the whole distributed PS data cube, sensor i should update its sensor value, prefix sum, and prefix average at approximately regular time intervals. Every time when sensor i has done so, it needs to increment its timestamp $ts(i)$, save $s(i)$ in its memory with $ts(i)$, and propagate the updated prefix sum to its neighbor.

For the other sensors, each sensor j needs to maintain a timestamp $ts(j)$ which is initially one. At any time t when any sensor j has received the prefix sum $s(k)$ from a neighboring sensor k, sensor j will save $s(k)$ in either one of $l(j)$, $u(j)$, or $d(j)$ as required by Algorithm 1 depending on the position of sensor k. If all of these three variables $l(j)$, $u(j)$, and $d(j)$ are updated, it can compute its prefix sum $s(j)$, and this $s(j)$ refers to the timestamp $ts(j)$. Hence

sensor j will save $s(j)$ and $ts(j)$ in its memory for future queries. Sensor j will then forward $s(j)$ to its neighbors at the bottom right of it.

Algorithm 4 shows the procedures for the sensors to maintain their prefix sums for the consistent retrieval of data aggregates. Notice that no sensor can update its sensor value deliberately, unless it has obtained all the prefix sums from its immediate neighbors at the upper left of it.

With Algorithm 4, data aggregates can be retrieved consistently from a distributed PS data cube. Recall that to obtain the aggregate sum or the aggregate average of any region in a distributed PS data cube, we need to retrieve the prefix sums from a set of target sensors. Now more than one version of prefix sums are stored in each sensor, therefore we have to specify the version of prefix sums we want by including a timestamp ts_Q in the query messages and retrieve the desired prefix sums with the timestamps equal to ts_Q from the target sensors. Finally, the resultant aggregate sum or aggregate average will be corresponding to logical time ts_Q.

If not all the sensors contain the prefix sums with the timestamps equal to ts_Q, then the query should be rejected because the required version of prefix sums are not yet ready. To resolve the problem, we can either re-send queries to the sensors with the timestamp ts_Q after a certain period of time, or specify another timestamp which is less than ts_Q. We should not calculate a aggregate sum or aggregate average using prefix sums with different timestamps, otherwise the resultant data aggregate will be inconsistent.

The prefix sums retrieved from the sensors are consistent because they have the same timestamp, and hence they correspond to the same logical time. This ensures that the prefix sums retrieved satisfy the second condition in Definition 12. It is because all the returned prefix sums must have the same timestamp. As a result, for any sensor i which is dependent on another sensor j, we can be convinced that the prefix sum $s(i)$ of sensor i will include the effect of the prefix sum $s(j)$ of sensor j.

Algorithm 4 Maintain Prefix Sum for Consistent Data Aggregate Retrieval

```
static ts(i) = 1
terminate = FALSE
l(i) = empty, u(i) = empty, d(i) = empty

if Sensor i is at (0,0) then
  Update v(i), s(i), and m(i) at regular time intervals
  Save s(i) with ts(i)
  ts(i) = ts(i) + 1
  terminate = TRUE
end if

if Sensor i is not at (0,0) then
  repeat
    if s(j) from sensor j is received then
      if j is on the left of i then
        l(i) = s(j)
      end if
      if j is on top of i then
        u(i) = s(j)
      end if
      if j is on the upper left of i then
        d(i) = s(j)
      end if
      if l(i), u(i) and d(i) are not empty then
        s(i) = v(i) + l(i) + u(i) − d(i)
        Save s(i) with ts(i)
        ts(i) = ts(i) + 1
        terminate = TRUE
      end if
    end if
  until terminate = TRUE
end if

if terminate = TRUE then
  i sends s(i) to j1 where j1 is on the right of i
  i sends s(i) to j2 where j2 is at the bottom of i
  i sends s(i) to j3 where j3 is at the bottom right of i
end if
```

Chapter 5

Conclusions

In this thesis the retrieval of data aggregates and the consistency of data in sensor networks are studied. For the retrieval of data aggregates in sensor networks, we propose to construct distributed prefix sum data cube and distributed local prefix sum data cube such that the fast and simultaneously retrieval of data aggregates from multiple regions in a sensor network can be realized. For the consistency of data in sensor networks, we proposed a synchronous concurrency control protocols for the proposed distributed data cubes so that they can be constructed, updated, and queried consistently.

In the literature many data aggregation techniques for sensor networks have been proposed, and these existing techniques are capable of returning the aggregate values of a single set of sensors. However, we notice that when data aggregates of several sets of sensors are needed at the same time, we have to build multiple distributed data structures or gossip groups in a sensor network. Hence in a sensor network with N sensors, we may need 2^N distributed data structures or gossip groups in order to get the data aggregates of all possible sets of sensors. To deal with the problem, we proposed to construct distributed prefix sum data cube and distributed LPS data cube in sensor networks. Using a distributed data cube, simultaneous aggregate sum and aggregate average queries on multiple regions in a sensor network can be answered in a

constant number of operations. We carried out simulations on the construction speed, network traffic, and scalability of the distributed data cubes. From our observations, the proposed techniques scale well.

In contrary to the retrieval of data aggregates, data consistency in sensor networks has rarely been mentioned. In the latter half of this thesis, we proposed a synchronous algorithm for the consistent construction and update of the proposed distributed data cubes. With this algorithm, data aggregates can be retrieved consistently from the two distributed data cubes.

As a future work, we strong recommend the consistency of data in sensor networks to be studied. Since sensor networks are application specific, it may be more desirable to decide different concurrency control protocols for different sensor network applications. With more sophisticated concurrency control protocols, we will be able to construct distributed data cubes in an asynchronous way.

Bibliography

[1] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors." *Commun. ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[2] D. Steere, A. Baptista, D. McNamee, C. Pu, and J. Walpole, "Research Challenges in Environmental Observation and Forecasting Systems," in *The 6th Annual International Conference on Mobile Computing and Networking (MobiCom 00)*, 2000, pp. 292–299.

[3] CORIE. Available: http://www.ccalmr.ogi.edu/CORIE

[4] ALERT. Available: http://www.alertsystems.org

[5] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," in *The ACM International Workshop on Wireless Sensor Networks and Applications WSNA '02*, September 2002.

[6] DUSKISL. Available: http://www.greatduckisland.net

[7] CARSENSE. Available: http://www.carsense.org

[8] T. G. Cleary and K. A. Notarianni, "Distributed Sensor Fire Detection," in *The International Conference on Automatic Fire Detection (AUBE 01)*, March 2001, pp. 139–150.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–54, 1997.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," in *The 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[11] Y. Yao and J. E. Gehrke, "Query Processing in Sensor Networks," in *In Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, January 2003.

[12] S. Madden, R. Szewczyk, M. Franklin, and D. Culler, "Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks," in *The 4th IEEE Workshop on Mobile Computing Systems & Applications*, June 2002.

[13] D. Kempe, A. Dobra, and J. E. Gehrke, "Computing Aggregate Information using Gossip," in *The 44th Annual IEEE Symposium on Foundations of Computer Science*, October 2003.

[14] I. Gupta, R. van Renesse, and K. Birman, "Scalable fault-tolerant aggregation in large process groups," in *The Conference on Dependable Systems and Networks*, 2001, pp. 433–442.

[15] R. van Renesse, K. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Managementand Data Mining," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 164–206, September 2003.

[16] N. Xu, "A Survey of Sensor Network Applications," *Survey Paper*, May 2003.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104. Available: http://citeseer.nj.nec.com/382595.html

[18] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization," U.C. Berkeley, Tech. Rep., 2001.

[19] uAMPS. Available: http://www-mtl.mit.edu/research/icsystems/uamps

[20] E. Welsh, W. Fish, and P. Frantz, "GNOMES: A Testbed for Low-Power Heterogeneous Wireless Sensor Networks," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, Bangkok, Thailand, May 2003.

[21] PC104. Available: http://www.isi.edu/scadds/pc104testbed/guideline.html

[22] Crossbow. Available: http://www.xbow.com/

[23] C. T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range Queries in OLAP Data Cubes," in *ACM SIGMOD 1997*, vol. 26, 1997, pp. 73–88.

[24] S. Geffner, D. Agrawal, A. E. Abbadi, and T. Smith, "Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes," in *The Int. Conf. Data Engineering, ICDE*, 1999.

[25] M. Riedewald, D. Agrawal, A. E. Abbadi, and R. Pajarola, "Space-Efficient Data Cubes for Dynamic Environments," in *The Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK)*, 2000, pp. 24–33.

[26] M. Riedewald, D. Agrawal, and A. E. Abbadi, "Flexible Data Cubes for Online Aggregation," in *The International Conference on Database Theory (ICDT)*, 2001, pp. 159–173.

[27] S. Geffner, D. Agrawal, and A. E. Abbadi, "The Dynamic Data Cube," in *The International Conference on Extending DataBase Technology (EDBT)*, 2000, pp. 237–253.

[28] S.-J. Chun, C.-W. Chung, and S.-L. Lee, "Space-efficient cubes for OLAP range-sum queries," *Decis. Support Syst.*, vol. 37, no. 1, pp. 83–102, 2004.

[29] Y. Xu, J. Heidemann, and D. Estrin, "Geography Informed Energy Conservation for Ad Hoc Routing," in *ACM MOBICOM'01*, July 2001.

[30] R. Finkelstein, "MDD: Database reaches the next dimension," *Database Programming and Design*, vol. 8, no. 4, pp. 27–38, April 1994.

[31] R. Agrawal, A. Gupta, and S. Sarawagi, "Modeling Multidimensional Databases," in *The 13th Int. Conf. Data Engineering, ICDE*, 1995.

[32] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson, "A Foundation for Capturing and Querying Complex Multidimensional Data," *Information Systems*, vol. 26, pp. 383–423, 2001.

[33] E. Codd, S. Codd, and C. Salley, "Providing OLAP (on-line analytical processing) to user- analysts: An IT mandate," *Technical report, E.F. Codd and Associates*, 1993.

[34] M. Stemm and R. H. Katz, "Measuring and reducing energy consumption of network interfaces in hand-held devices," *IEICE Transactions on Communications*, vol. E80-B, no. 8, pp. 1125–31, 1997. Available: citeseer.ist.psu.edu/stemm97measuring.html

[35] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," in *Mobile Computing and Networking*, 2001, pp. 85–96. Available: citeseer.nj.nec.com/chen02span.html

[36] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.

[37] R. E. Stearns, P. M. L. II, and D. J. Rosenkrantz, "Concurrency control for database systems," in *FOCS*, 1976, pp. 19–32.

[38] C. H. Papadimitriou, "The serializability of concurrent database updates." *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[39] R. H. Thomas, "A solution to the concurrency control problem for multiple copy databases," in *Proceedings of the IEEE COMPCON*, 1978.

[40] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking.* ACM Press, 2000, pp. 56–67.

[41] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, "Building efficient wireless sensor networks with low-level naming," in *Symposium on Operating Systems Principles*, 2001, pp. 146–159. Available: citeseer.nj.nec.com/heidemann01building.html

[42] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proceedings of the 22nd International Conference on Distributed Computing Systems.* Vienna, Austria: IEEE, July 2002, p. 457.

[43] C. de M. Cordeiro and D. P. Agrawal, "Mobile Ad Hoc Networking," *In 20th Brazilian Symposium on Computer Networks*, pp. 125–186, May 2002.

[44] L. H. Lee and M. H. Wong, "Aggregate sum retrieval in sensor network by distributed prefix sum data cube." in *AINA*. IEEE Computer Society, 2005, pp. 331–336.

[45] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating Aggregates on a Peer-to-Peer Network," *Technical Report, Computer Science Department, Stanford University*, 2003.

[46] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, June 1981.

[47] D. P. Reed, "Naming and synchronization in a decentralized computer system," *Technical Report: TR-205, Massachusetts Institute of Technology*, 1978.

[48] R. E. Stearns and D. J. Rosenkrantz, "Distributed database concurrency controls using before-values," in *ACM SIGMOD 1981*, 1981, pp. 74–83.

[49] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries, "The Implementation of an Integrated Concurrency Control and Recovery Scheme," in *ACM SIGMOD 1982*, 1982, pp. 184–191.

[50] A. Chan and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Trans. Software Eng.*, vol. 11, no. 2, pp. 205–212, 1985.

[51] W. E. Weihl, "Distributed version management for read-only actions," *IEEE Trans. Software Eng.*, vol. 13, no. 1, pp. 55–64, January 1987.

[52] D. Agrawal and S. Sengupta, "Modular Synchronization in Distributed, Multiversion Databases: Version Control and Concurrency Control," *IEEE Trans. Knowledge Data Eng.*, vol. 5, no. 1, pp. 126–137, February 1993.

[53] O. T. Satyanarayanan and D. Agrawal, "Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases," *IEEE Trans. Knowledge Data Eng.*, vol. 5, no. 5, pp. 859–871, October 1993.

Appendix A

Publications

- **LEE Lok Hang** and WONG Man Hon. Aggregate Sum Retrieval in Sensor Network by Distributed Prefix Sum Data Cube. In Proceedings of The 19th International Conference on Advanced Information Networking and Applications (AINA2005). Taipei, Taiwan, March 2005. [44]

- **LEE Lok Hang** and WONG Man Hon. Data Aggregation in Sensor Network by Distributed Data Cube. Submitted to IEEE/ACM Transactions on Networking.

- **LEE Lok Hang** and WONG Man Hon. Simultaneous Aggregate Sum Retrieval from Multiple Regions in Sensor Networks by Distributed Data Cubes. Submitted to International Journal of Wireless and Mobile Computing.